日本コンピュータ協会編/コンピュータ・サイエンス研究書シリーズ ③

オペレーティング・システムへの構造的アプローチ（上巻）

日本アイ・ビー・エム㈱
江村潤朗 著

科学技術出版社

コンピュータ・サイエンス研究書シリーズ

オペレーティング・システムへの構造的アプローチ（上巻）

日本アイ・ビー・エム㈱　江村潤朗 著

科学技術出版社

ま え が き

オペレーティング・システムは，その名前が示す通りコンピュータ・ハードウェアを動かすための“システム”です．ハードウェアを動かすという以上，それはプログラムであるということになります．単なるプログラムではなく，コンピュータ・ハードウェアの生産性を向上させるという目標のために種々のプログラムを有機的に結合したものです．しかも，オペレーティング・システムは，完成しきった固定的システムではありません．コンピュータ・ハードウェアの技術革新，コンピュータ利用の方向あるいはコンピュータが置かれているシステム環境を考慮に入れながら，絶えず変化し，発展し，拡大してゆく性格をもったシステムです．

そもそも，われわれが一般に対象としている“システム”は目標ないしは目的に依存して機能しています．何らかの目標を達成するために沢山のシステム構成要素が相互に関係し合いながら機能しており，システムの構成要素が増えれば増えるだけ，他の要素との相互依存関係は複雑化します．その結果，ある一つの特定要素が何のために機能しているのか定かでなくなってしまうのです．したがって，その要素だけを抽出してきて検討してみても，その存在意義を明確にできないのが普通であり，そのシステムがねらいとしていることから出発し，そのシステムの目標を達成するために関連している各要素の全体的姿を把えたうえで，各要素がいかに機能し合っているかを見極めてゆく必要があります．マクロからミクロへとシステム構造を解明してゆくやり方です．このようなアプローチも，一種のシステムズ・アプローチと呼べるかも知れませんが，特に私自身は“構造的アプローチ”と呼んでいます．対象としているシステムを1つの構造として把え，それを構成する要素を有機的に位置づけてゆくやり方は，単に対象が理解しやすいだけでなく，対象が変化してゆく場合にその変化を容易に意味づけてゆくことにも役立ちます．たとえば，システムにある新しい機能が付加されたとします．その機能がどういう意味を持つかによって，構造化したシステムの中に容易に組込んで考えてゆくことができます．つまり，変化を構造の中に位置づけて考えることができるのです．そこで，本書においてはオペレーティング・システムを理解してもらうためのアプローチと

して“構造的アプローチ”を採用してみました.

本書では，まず第1章でこの構造的アプローチを明確にしたうえで，オペレーティング・システムの“システム”としての意義づけを行いました.

第2章の「オペレーティング・システムの概念的構造」では，オペレーティング・システムの出発点ともいうべき設計目標を明確化し，オペレーティング・システムの全体的機能構造を解明するように心がけました.

第3章では，「オペレーティング・システムのためのハードウェア構造」と銘うって，オペレーティング・システムの設計フィロソフィーを十分生かすために，ハードウェア構造に組込まれてきた主要機能の意味づけを試みました.

第4章「プログラム設計構造と連係編集プログラム」では，オペレーティング・システムのもとで，プログラムの設計手法としてどのような方法が工夫されており，かつ，それらが制御プログラムのもとでどのように実行されるかを言及して，さらに，プログラムの結合機能を持つ連係編集プログラムの構造と機能の解明をしました.

第5章「データ管理の諸機能とその構造」では，オペレーティング・システムを構成する制御プログラムの三大機能の1つであるデータ管理機能を取りあげて，その機能と構造について説明しています.

なお，ジョブ管理の機能とタスク管理の機能に関しては下巻で取扱います.下巻では，多重プロミラミングや仮想記憶の機能などについても説明する予定です.

出来るだけ構造的に理解してもらうことに主眼をおいていることから，視覚的に把握してもらうための図版を随所に挿入しました.上巻だけでも100枚以上の図版を入れました.なお，本書の内容は，雑誌**「ソフトウェア技術」**（日本コンピュータ協会・発行）の第2巻第2号以降に十数回にわたって連載した内容に手を加えたものであり，上巻の後半および下巻の全部はまったく新たに書き下したものです.

最後に，本書の執筆にさいして再三の執筆中絶にもかかわらず，終始激励しご協力くださり，発刊にこぎつけられました日本コンピュータ協会の村越勝弘編集長をはじめとしたスタッフの方々に感謝の意を表したいと思います.

昭和47年4月

著　者

目　　次

第 1 章

オペレーティング・システムの システム的構造

一．はじめに

コンピューターのソフトウェア面を代表する言葉として **“オペレーティング・システム** (operating system)” なる用語が一般化したのは第三世代のコンピューターからである．もちろん，第二世代においても大型機種にはオペレーティング・システムがつきものであったが，いまだ一般的ではなかった．

オペレーティング・システムという体系化されたプログラム群のもとでコンピューティング・システム（ハードウェア群）を作動させることによって，システムとしての生産性を高め，同時にプログラマーやオペレーターの負担を軽減するという基本目標は第二世代も第三世代も変りはない．コンピューティング・システムそのものの処理速度が比較的遅く，コンピューターで処理する仕事自体も単純である場合には，コンピューターで仕事を処理している過程でかなり無駄があってもさほど問題視されなかったし，プログラム作成に負担がかかっても，解決すべき問題が複雑でないだけにさほど気にもならなかった．しかし，コンピューティング・システムの処理速度が高速化し，仕事も高度化するにつれて，これまでないがしろになっていた領域が問題となってきた．とりわけ，科学技術計算のように１つの仕事がせいぜい数分で片づく場合には深刻な問題となってきた．ある問題を処理するために磁気テープやカード・デックのような入力媒体を用意したり，プログラムをローディングするといった準備に数分間を要していたのではコンピューターの処理能力は半減してしまう．そ

れに解決すべき問題自体が高度化し，単位時間で処理されるプログラム数が増加してくるとなるとプログラム作成も単純化されないと困ることになる．手工業的コンピューターの利用から機械工業的なコンピューター活用の方向に脱皮せざるを得なくなってくる．

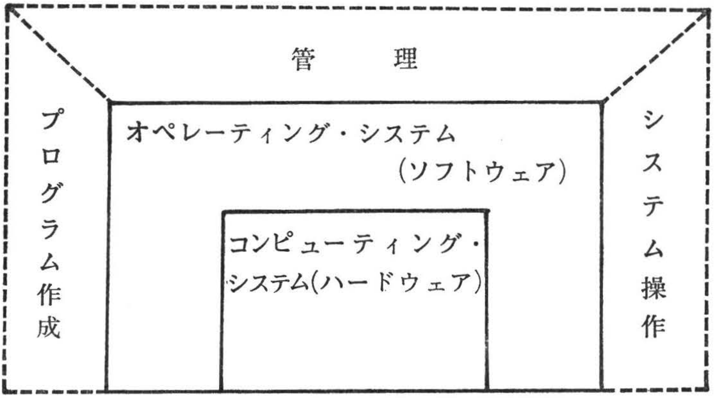
つまり，組織的なプログラム体系のもとでコンピューターを無駄なく作動させるとともに，プログラム作成や機械操作の単純化を一連のソフトウェアでおこなってゆこうということにつながる．

換言すると，すべてのプログラムと処理すべきデータを一定の上位プログラム（一般に制御プログラムと呼ばれている）群に管理させることによってコンピューター処理過程を最大限に自動化させ，コンピューターそのものの生産性を高め同時にオペレーターやプログラマーあるいはコンピューター部門の管理者の非生産的時間を短縮することにより，コンピューター部門全体の効率を高めてゆこうという発想にもとづいたソフトウェア体系が，オペレーティング・システムである．

コンピューターの速度が驚異的に速くなり，その利用が拡大化しかつ複雑化してくると，前述の問題は単に科学計算についていえるだけでなく事務計算や経営計算にも十分あてはまるようになった．こうなってくると小型のコンピューターは除外するとしても中型以上のコンピューターでは，その性能を発揮させシステム全体の生産性を高かめるためにオペレーティング・システム的なソフトウェアへのアプローチが必然となってきた．これが第三世代である．

第三世代のオペレーティング・システムは，ユーザーの作成した**問題処理プログラム**，FORTRAN や COBOL コンパイラーのような**言語翻訳プログラム**(language translator) およびユーティリテイー・プログラムや分類／組合せプログラムのような**サービス・プログラム**などからなる**処理プログラム**（processing program）と，それらのプログラムの実行過程を管理しコンピューティング・システムを最大限に活用するように仕事を連続・併行的に処理するための**制御プログラム**(control program)からなる統合化されたプログラムの集りである．このオペレーティング・システムがコンピューティング・システムを自己の支配下におくことによって，コンピューティング・システムの性能と機

能を拡充する．**第1図**に示すように，両者は一体となってシステムを形成し，そのシステムの首尾一貫した管理と，プログラムの作成および運用が行なわれることになる．



第1図　第三世代のシステム構成

第三世代のオペレーティング・システムといっても，具備している機能の度合いあるいは機械の構成装置によっていくつかのレベルが存在するし，それぞれのメーカーが提供しているオペレーティング・システムによってもかなりの差がある．とはいうものの，その基本目標としているところはいづれも同じであるのでオペレーティング・システムの基本構成要素や機能に大差はない．それゆえに，本書では全てのオペレーティング・システムをとりあげて論ずるつもりもないし，特定オペレーティング・システムについて詳細に説明し，そのオペレーティング・システムを自由自在に使いこなしてもらうつもりもない．

オペレーティング・システムの基本となる構造と機能を把握していただくことによって特定オペレーティング・システムへの接近の手がかりとしていただくと同時に来るべき未来のオペレーティング・システムを洞察する手がかりを提供することを目標としている．ただし，論旨の焦点を明確化し，具体性をもたせるための便宜上ＩＢＭシステム/360オペレーティング・システムの基本構造を中心に進めてゆくことにする．

二．構造的アプローチのねらい

かりに，読者が特定オペレーティング・システムを利用することに急を要するのであれば，即刻そのシステムだけに焦点をしぼった学習をされることを期待する．そうでないのならオペレーティング・システムへの構造的アプローチをまずすすめたい．

というのは，コンピューターに関係するハードウェア，ソフトウェアあるいはその利用技術の革新は極めて激しいからである．もし，今読者が特定オペレーティング・システムについてその使い方を修得したとしても，1，2年内に，その知識と技術のほとんどが役に立たなくなるおそれがある．中途半端な学習方法でその使い方だけをマスターしたのではその耐用年数は極めて短かいものと云えよう．しかも，これまでのコンピューター教育がとかく対象を平面的に把え知識を注入する傾向が強かったため，創造的能力開発に弱くなっており，何か新しい概念のソフトウェアなり利用法が出現するとそれを過去の技術や手法と断続したものとして理解せざるを得ないような頭脳構造にしたて上げてしまっているようだ．だから，少しでも新しい機能や概念をもったオペレーティング・システムが出てくるとその修得に四苦八苦することになり，あたかも，長年の蓄積が無用になってしまったかのごとき状況を呈することになってしまう．変化に対して極めて効率の悪い対処の仕方をしているのである．

では一体，変化に強い学習方法はどのようなものであろうか．それは，ある概念とか事実の表面的知識を得る方向ではなくて，その基底にあるもの，つまりその構造を重視するアプローチである．表面的知識や利用法を修得することに努力するのではなく，知識や事実を再構成して新しい概念や手法にあてはめうるのに必要となる構造を理解することである．たとえば，ある新しい機能がオペレーティング・システムに追加されたとすると，オペレーティング・システムの基本構造を充分把握していればその構造の中に新しい機能を適格に位置づけて理解できる．もしも，新しい概念のオペレーティング・システムが出現したとしても，それが本来的に目標としているものに大きな変化がある筈はな

いので，従来のシステムを構成していた要素の大半は何らかの形で包含されている．したがって，従来の構造を再構成する能力が身についているならば，新しい概念を容易に了解し得るのである．

昔から“木を見て森を見ず”ということがよく言われる．森全体の構造を充分に把握したうえで，部分，部分の集合を位置づけ，さらに１本１本の木を位置づけなければ，森の中に迷い込んでしまって，新しい森にぶつかったときそれが森としてとらえられなくなってしまうのである．つまり，学習の転移が働かなくなってしまう．英語のもつ基本構造を適切に修得した人は，それに近い構造を持ったフランス語やドイツ語を容易にマスターするという．つまり，応用性が効くようになっているわけで，これをプラスの転移という．一方，生半可に英語を理解している人は，たとえその人がペラペラ英語が話せたとしても，フランス語を修得するさいには，生半可な英語の知識がかえってさまたげになる．これがマイナスの転移である．このことはコンピューターについて学習する場合にもあてはまる学習原理である．読者が現在あるオペレーティング・システムを使いこなせたとしても，その基本構造をしっかり理解していないならば新しい概念のオペレーティング・システムに直面したとき，現存のオペレーティング・システムに全く無知な人にも遅れてしまう可能性がある．したがって，一見まわり道のようではあるがまっとうなオペレーティング・システムの学習方法は，まずそれに構造的にアプローチしてゆくことである．

今後は，これまで以上に技術の革新が加速化してゆくことは疑いないし，われわれが修得すべき情報量もそれにしたがって幾何級数的に増加するであろう．このような状況を想定した場合，このままでは結局情報の渦におし流されてしまう．したがって，ダイナミックに変化する状況を機敏に把え，そのときどきに生起する新しい情報や概念を適切に把握して，既習知識の中に正しく位置づけたり，逆に既習知識を新しい概念の中で適切に再構成してゆくことのできる変化に強いアプローチこそ重要である．

本書では，まずオペレーティング・システム全体の基本構造にかかわる目的，主要構成要素とそれぞれの諸機能を把えたうえで諸要素の相互依存性を解明しそのあとで個々の主要プログラムを構成する要素の機能と関連性を追求し

つつ，たえず全体構造の中での位置づけをおこなってゆく．

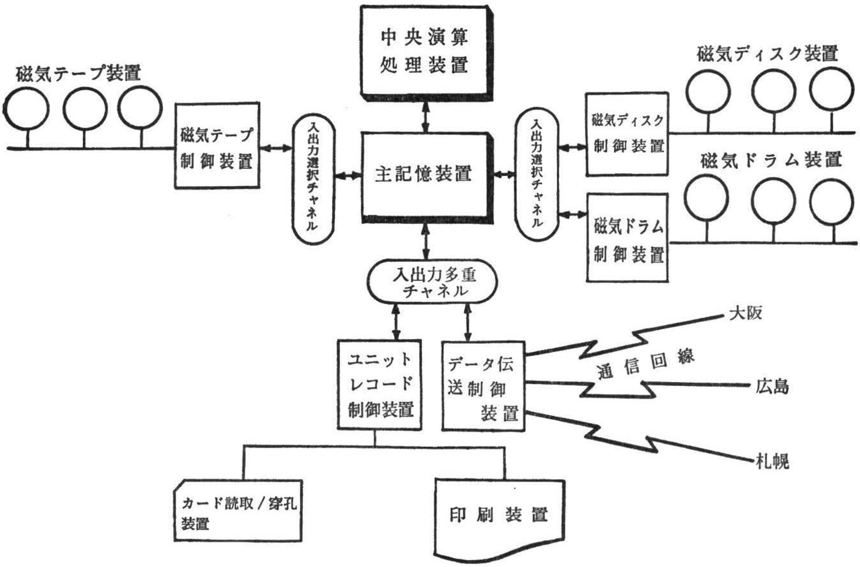
三．“システム”としてのオペレーティング・システム

1.　一般的システムの特性

われわれの周囲にはシステムと呼ばれるものが沢山存在している．自分自身の生体そのものが一つのシステムでもありうるし，われわれが働いている企業そのものがシステムでもある．さらにわれわれがコンピューター部門に属しているとするなら，その部門自体がまたシステムでもありうる．もっとも企業というシステムで把えた場合には，それはサブ・システムという存在になる．逆にコンピューター部門を一つのシステムと想定した場合には企業はシステム環境として作用をおよぼすようになる．このように問題とする領域をどう設定するかによって，システムは大きくもなるし，小さくもなる．また，システムを構成する内容にもちがいが生じてくることにもなる．

このようなシステムと呼ばれるものは，一つの構成要素からなりたっていることはない．必ずいくつかのサブ・システムや要素からなりたっており，ある特定の目的や計画を遂行するためにその諸要素が秩序立った統一体を構成している筈である．無目的に集まった烏合の衆はシステムとはなりえないのである．共通の目的に奉仕するために有機的関係をもった構成員の集まりでなければシステムとは呼べない．つまり，システムを構成する要素は，各要素がばらばらに機能しているのではなく相互に作用をおよぼしたり，あるいは依存関係を保ちながら特定の目的や計画に奉仕している行動や機械，物事の仕組や体系である．しかもシステムは，何らかのものを入力として受け入れ，それをシステム内で処理して，その結果を出力するという性質もある．

たとえば，**第2図**に示したのがコンピューターのハードウェア構成である．一般に**コンピューティング・システム**（computing system）という言葉で表現されるようにこれも立派な“システム”である．入力装置から読みとったデータをプログラムの指示にもとづいて迅速に処理してゆくという目的のために，複



第2図　コンピューテイング・システムの基本構成

数個の装置から構成されたシステムである．それぞれの装置は自己に課された機能を果すと同時に相互に有機的に機能し合いながら作動しているのである．

特定の指示にもとづいて，まずプログラムが入力装置から制御装置とチャネルを経由して主記憶装置に記憶される．記憶されたプログラムの最初に実行すべき命令が中央演算処理装置に取り出されたうえで，何を行うべき命令かが解読される．もし，その命令が入出力命令であったなら，チャネルや入出力装置の状態を調べ，指示されたチャネルと入出力装置が利用可能状態であれば，入出力処理の制御権をチャネルに移譲し，中央演算処理装置は次の命令の解読に移る．中央演算処理装置の処理に併行してチャネルは入出力制御の任にあたる．チャネルが引き受けたプログラム（これを**チャネル・プログラム**という）を構成する命令の中に特定入出力装置固有の仕事，たとえば磁気テープの巻き戻しといったような，磁気ディスクや印刷装置のような他の装置では行わない機能に関する仕事がある場合にはその役割を磁気テープ制御装置に引きわた

す．この制御装置は，指示された磁気テープ装置のテープの巻き戻しを行う．もし，チャネル・プログラムの中の命令が“読め”という命令なら，チャネルは指示された装置から特定桁数のデータを入力させる．入力データは制御装置とチャネルを経由して主記憶装置に貯えられる．無事全桁読み込んだなら，それは入力に関する割込みという方法で入力完了の旨が中央演算処理装置に伝えられる．これによって，入力データの処理が中央演算装置で行われるようになる．

このように各々の装置が実に有機的に機能し合いながらデータの読み込み，処理，結果の出力といった仕事を完遂してゆくのがコンピューティング・システムである．

以上みてきたように，一般に**システム**と呼ばれるものの特性はつぎの諸点にまとめ上げることができる．

(1)　特定の目的あるいは計画をもっていること．

(2)　複数個の構成要素からなりたっていること．

(3)　それぞれの構成要素は相互関連性をもちながら機能すること．

(4)　何らかの入出力があること．

(5)　システムは特定の環境（システム環境）があってはじめて機能すること．

これが一般的システムの特性であるが，一体オペレーティング・システムとはどのようなシステムであろうか．

2.　オペレーティング・“システム”

かってオペレーティング・システムの概念がソフトウェアに導入されていなかった時代には，個々のプログラムが全く独立的にかつ完全に機能するものとして作成された．記号言語で書き上げた**原始プログラム** (source program) を**機械語** (machine language) に翻訳すればそれで完結したプログラムとなったのである．ただ，機械語プログラム・デックの先頭に**ローダー** (loader) と呼ばれる数枚のカードを置くことによってプログラムが主記憶装置の所定の場所に貯えられるようにすればよかった．そのプログラムによる処理が終了すれ

ば，別の仕事のプログラムを同様の手順で主記憶装置に記憶させて処理をつづけてゆく．このような事態にあっては，いくつかのプログラムが相互に関連性をもったシステムを形成する必要は毛頭なかったのである．

ところが，先にふれたように技術の進歩によってコンピューターの内部処理速度が高まってくると，このような一発処理方式の限界がでてきた．というのは，内部速度がいかに高速になったとしても，人間が手操作で遂行する仕事，つまり，特定の仕事の処理のために磁気テープや磁気ディスクを準備したりその仕事のためのプログラム・デックを読み込ませたりの仕事に要する時間はほとんど一定である．技術進歩がこのような手作業を軽減することにほとんど貢献しないのである．したがって，コンピューターの速度が高まれば高まるだけ手操作のために失ってしまう時間が全体のデータ処理時間の割合の多くを占めてしまうことになる．いいかえれば，コンピューターが実際に処理している時間に較べて処理体制を整えるための準備時間（この時間はコンピューターは小休止しているのだが）の割合が相対的に増えることになる．

さらに，コンピューターが大型化し，処理速度があがってくると当然それに見合った高度な適用業務を処理対象とするようになるので，必然的に主記憶装置の容量は大きくなり，手足となる入出力装置も多様化すると同時に沢山の装置が一台のコンピューターに接続されることになる．こうなると，コンピューターそのものが何かを処理しているときでも，多くの構成装置が遊んでいる可能性は高くなってくる．ある特定業務を処理している最中に何台かの入出力装置はその処理に関係しないため遊んでいるかも知れないし，その業務に使うプログラムはあまり大きくないために主記憶装置の大半は無駄になっているかも知れない．これでは，コンピューティング・システムがシステムとして最適活用されているとはいえない．システム全体としては極めて効率の悪い使い方である．

この２点が主要な理由となって，コンピューティング・システムを効果的に運転してゆくためのシステム的アプローチの必要性がでてきた．

1）オペレーティング・システムの根源的目標

これまでのことから予想されるようにオペレーティング・システムの根源的目標は，多種多様な適用業務を首尾一貫したソフトウェア・システムのもとで処理することによってコンピューティング・システム全体の生産性を最大限に発揮させようということである．したがって，コンピューティング・システムを動かす全てのプログラムがオペレーティング・システムを構成する要素となって，そのシステムのもとで働くことになるのである．つまり，オペレーティング・システムは，コンピューター全体の効率を高めるために各種プログラムが有機的に統合されたものである．

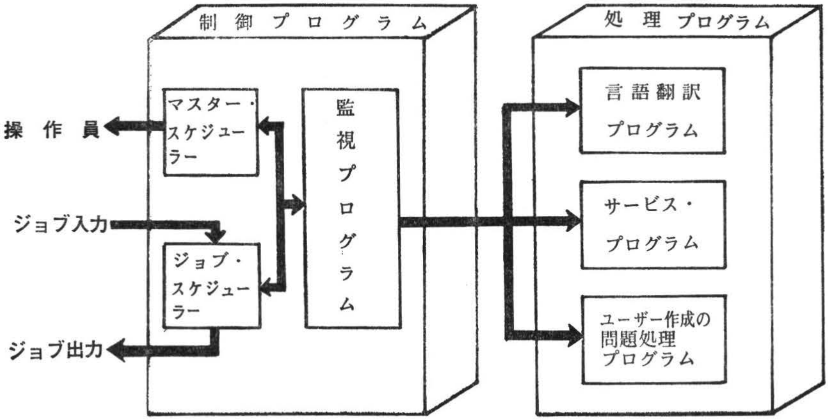
2)　システムを構成する要素

第3図に示しているようにオペレーティング・システムを構成する二大要素は制御プログラムと処理プログラムである．**制御プログラム**（control program）は，自らは何ら生産的な仕事を行わないが処理プログラムの実行過程やコンピューティング・システムの作動状態を監督する役目をもっているし，次に処理すべき仕事の準備やプログラムの準備を行う．一方，**処理プログラム**（processing program）は制御プログラムの監視のもとで実際に何らかの生産的仕事に従事するプログラムであり，われわれが普通**プログラム**と呼んでいるものである．

処理プログラムは，メーカーが提供するプログラムとユーザーが作成するプログラムに大別される．われわれが書いたプログラムを機械語に変換する**言語翻訳プログラム**（コンパイラーやアセンブラー）と分類／組合せプログラム（ソート・プログラム）のような**サービス・プログラム**（service program）と呼ばれるものはメーカーが提供する処理プログラムである．さらに，ＬＰとかシミュレーション・プログラムのような汎用適用業務プログラムもメーカーが提供する処理プログラムである．

制御プログラムを大別すると，絶えず処理プログラムの実行過程とコンピューティング・システムの作動状態を監視する役目を果す**監視プログラム**（supervisor），操作員とのコミュニケーションの仲介の任にあたる**マスター・スケジューラー**（master scheduler），ある仕事（ジョブ）から

他の仕事へ自動的に移行するための準備と仕事の後始末を行う**ジョブ・スケジューラー**（job scheduler）の3つのプログラムになる．これらプログラムの詳細は後述する．



第3図　オペレーテイング・システムの全体的構造

3)　構成要素の相互関連性

第3図で示したように制御プログラムの機能は大別するとつぎの3つになる．まず第1は，仕事を連続的に処理してゆくためにある仕事が終了したときその後始末（たとえば，不用になった入出力媒体の解放）を行ったうえで，次に処理すべき仕事の段取りを行う．その仕事に必要とする入出力装置を割り当てたり，プログラムの実行開始の準備をしたりといったことなどで，この機能を果すのがジョブ・スケジューラーであった．実行すべきプログラムの準備が終了すると，あとは監視プログラムの監督のもとでそのプログラムが実行される．この処理プログラムの実行過程において，プログラム上のミスや機械の故障が発見されるとそれに対する定められた処理を行う．あるいは入出力操作の開始の指示や入出力操作中の監視も行うし，また同時に複数個の仕事を1つのコンピューターで処理してゆく（このことを一般に**多重プログラミング**と呼んでいるが）過程での中央演算処理装置の占有権の制御なども行う．これが監視プログラムの働きで

ある．もし，監視プログラムが何か例外的事態を見つけたり，処理プログラムの依頼によって操作員に何らかの行動をとらせたいときには，マスター・スケジューラーを通して操作員にメッセージが出される．一方，操作員がシステムに対して何らかの要求や指示を行うとそれらはマスター・スケジューラーを通して所定の処理が行われる．つまり，システムと操作員とのコミュニケーションを担当するのがマスター・スケジューラーであった．

これら制御プログラムを構成する3つのプログラムの管理下で実際の生産的仕事を遂行するのが**処理プログラム**である．この関係はちょうどわれわれの人間組織と同じである．ある組織の管理者は，自らは何ら生産的な仕事に従事しないが，部下となる人々や割り当てられた予算とか施設といった資源の最適管理を行いながら，部下に実際の生産的仕事を行わせている状態を想起していただきたい．その部門で行う仕事は関連する部門から入力として管理者が受けとり，仕事に必要となるスケジュールを立てたうえで適切な部下（処理プログラムに当る）にその仕事をやらせる．自らはその進行状況を監督しつつ，特殊な場合や部下の質問などの処理を行う．あるいは部下が誤った処理をしていると思われるものを摘出して改善指導を行なう．時には他部門とかけ合う必要もでてくるし，逆に他部門からの要請にこたえる必要もある（マスター・スケジューラーの機能に相当）．ある仕事が完遂されると，担当者をその仕事から解放し完成した仕事は関連部門に出力として提供する．このように管理者はオペレーティング・システムの制御プログラムと同じように自らは何ら生産的仕事は行わないが，持てる資源を最適活用して全体としての業積を上げるようにするのが本来の姿であろう．

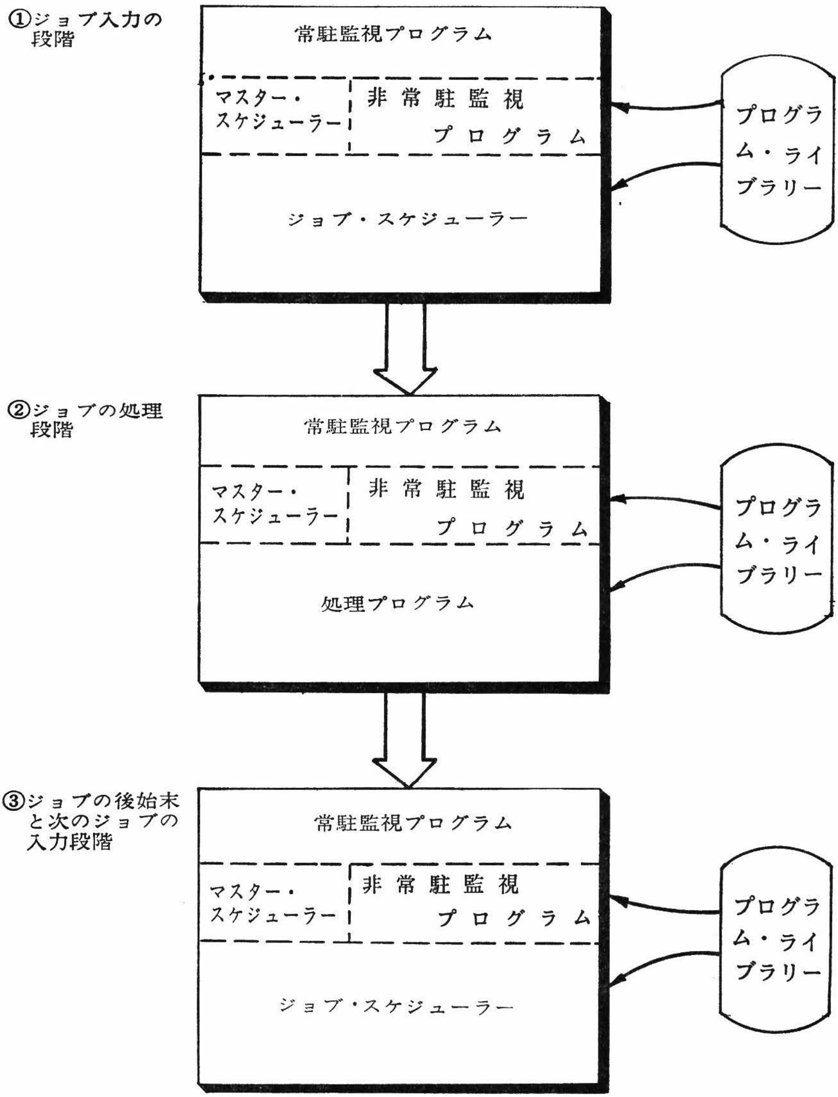
オペレーティング・システムは制御プログラムだけでは何ら意味をなさないし，逆に処理プログラムだけあっても何ら生産できない．両者が一体となってコンピューティング・システムに作用したときはじめてその機能を果すことができるのである．ここで，オペレーティング・システムがコンピューティング・システムのもとでどのように働くかについて若干ふれ

ておこう．

第4図をみていただきたい．オペレーティング・システムを構成する幾多のプログラム群は一体どこに収容されているのであろうか．オペレーティング・システムを構成するプログラム群は，磁気ディスクや磁気ドラムあるいは磁気テープ（最近は磁気テープは逐次処理しかできないことから敬遠されてはいるが）上の**プログラム・ライブラリー**（program library）に記憶されている．そのうち，そのときどきに必要となるプログラムが主記憶装置に呼び込まれて実行される．ただし，監視プログラムのうちで使用頻度の高いルーチンは**常駐監視プログラム**（resident supervisor）として，コンピューターが作動しているかぎり常に主記憶装置におかれる．操作員とのコミュニケーションに当るマスター・スケジューラーも頻繁に利用されるので常駐させる場合が多い．一方，あまり使用頻度の高くない監視プログラムのルーチン，たとえばエラーの処理ルーチンのようなものは，その処理が必要になる都度プログラム・ライブラリーから**非常駐監視プログラム**（transient supervisor）の記憶域に呼び込まれて，当該処理を行なう．別の例外処理が生じると非常駐監視プログラム域に入っている前の例外ルーチンにオーバレイする方法で呼びこまれる．こうすることによって，主記憶装置の経済性を高めている．

制御プログラムのうちでもある仕事と次の仕事の橋渡しのためだけに必要となるジョブ・スケジューラーは問題処理プログラムが記憶される領域を共同利用することになる．つぎの仕事に移る準備を行っている段階にはまだ処理プログラムが主記憶装置に呼び込まれていなくても一向にさしつかえないからである．一たん準備が終るとその仕事が完了するまでジョブ・スケジューラーの機能する場面はないので，ジョブ・スケジューラーが呼び込まれていた同じ記憶域に処理プログラムが記憶されるわけである．

このような方法で監視プログラムと処理プログラムあるいはジョブ・スケジューラーが同時に主記憶装置に記憶されている状態で相互が有機的な連係を保って連続的に仕事を処理してゆくのである．



第4図　主記憶装置のプログラムの相互関連図

4) ジョブ入力とジョブ出力

オペレーティング・システムへの入力は仕事単位に行われる．仕事の名前，その仕事で使用する入出力ファイルとその性質，その仕事を達成するためのプログラムの名前等が入力情報である．この仕事の単位のことを**ジョブ**（job）と呼んでいる．仕事を達成するために必要となる入力データはあくまで処理プログラムへの入力であってジョブ入力ではない．

処理プログラムが実行されたあと，その仕事がどういう状態で終了したのか，どれぐらいの時間を要して終了したのか，入出力装置の後始末はどのようにすべきかの情報がジョブ出力である．処理プログラムの実行過程での演算結果の出所はジョブ出力ではなくその処理プログラムの出力である．

つまり，オペレーティング・システムというシステムでとらえた場合の入出力単位はあくまで**ジョブ**という単位であり，オペレーティング・システムを構成する一要素である処理プログラムへの入出力単位は**データ・レコード**という単位である．

5) オペレーティング・システムの環境

オペレーテーング・システムはコンピューティング・システムという環境のもとではじめて機能することができる．このコンピューティング・システムがオペレーティング・システムの内的環境である．オペレーティング・システムの内的環境を形成するコンピューティング・システムの構成装置は多種多様である．第2図に示したコンピューティング・システムの基本構成のあるものが欠ける場合もあるし，逆にここで示した以外の装置がとりつけられることもあるだろう．それに各々の装置に特殊な機構が付加されることだってある．すなわち，それぞれのコンピューティング・システムの機械構成は異るといってよい．

一方，コンピューターを実際に動かしているのは人間の集まりである．コンピューター部門の管理者，システム分析者，プログラマーあるいは操作員たちである．さらには，企業全体がこの集団の行動を規制している．

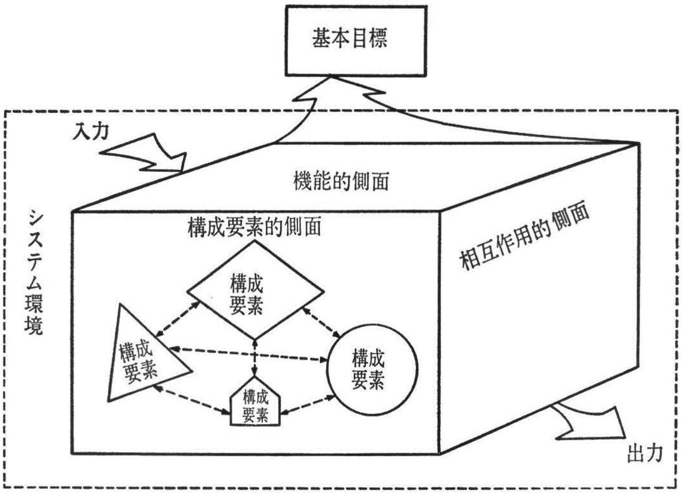
これらは，オペレーティング・システムの外的環境といってよかろう．これらの外的環境がコンピューターの利用方法や利用分野を決定している．したがって，その方法や分野は当然企業ごとに異ってくる．

機械構成が異り，適用方法や分野もことなると当然システムとして要求されるオペレーティング・システムの構成要素や内容に差異が生じる．それぞれの要件に適合するような全てのプログラムをあらゆるオペレーティング・システムに包含させるとなるとオペレーティング・システムの内容はふくれ上り，ひいては非常に効率が悪いものとなってしまう恐れがある．

これに対拠するためにオペレーティング・システムはどう融通性を効かせているのであろうか．現在のオペレーティング・システムにおいては，システム環境に対しての充分な考慮が払われている．オペレーティング・システムの根源目的があらゆる適用に対して最大の生産性を上げることを指向していることからすれば当然である．オペレーティング・システムは既製品として提供されるものではなく，各機械構成と適用業務の要件に合わせて仕立て上げができるシステムである．この仕立て上げのことを一般に“**システムの生成**（system generation）”と呼んでいる．オペレーティング・システムのスケルトンはメーカーが提供するがこれを容易に最適システムに仕立て上げることのできるプログラムもオペレーティング・システムの一要素としてくみ込まれている．

これまで論じてきたようにオペレーティング・システムはシステムとしての一般的特性を備えているばかりでなく，システムとして完全なものへと絶えず成長をつづけている．オペレーティング・システムを固定的なものとして把握し利用することの危険性をいまさら強調することもあるまい．絶えず発展してゆくものとして構造的にアプローチしてゆくことこそが変化に対拠してゆくための必要条件である．

以上オペレーティング・システムのシステム特性を極めて大雑把に概観してきた．これでオペレーティング・システムのシステムとしての輪郭がある程度つかめてきたと思う．**第5図**にシステムとしての特性をまとめて構造化してお



第5図　システムのまとめ

いたのでこの構造に合わせてオペレーティング・システムの構造をあとづけしていただきたい.

第 2 章

オペレーティング・システムの概念的構造

一．情報処理システムを構成する三大要素

前章では，オペレーティング・システムの“システム”としての特性に焦点を合わせて，その構造を概観した．その結果，オペレーティング・システムの特性は，５つに集約されることがわかった．

⑴ 科学計算，事務計算あるいはテレ・プロセシングといったあらゆる適用業務を体系化されたソフトウエアのもとで処理することによって，コンピューター全体の生産性を最大限に発揮するという根源的目標をもっている．

⑵ オペレーティング・システムは，制御プログラムと処理プログラムから成り立っている．制御プログラムは，監視プログラム，ジョブ・スケジューラー，マスター・スケジューラーといったプログラム群に大別される．一方，処理プログラムには，言語翻訳プログラム，サービス・プログラムおよびユーザー作成のあらゆる問題処理プログラムが含まれている．

このようにオペレーティング・システムは，多種多様なプログラムをその構成要素としている．

⑶ 制御プログラムは，それ自体何ら生産的な問題処理にはたずさわらないが，処理プログラムの実行過程やコンピューティング・システムの作動状態を監督し，必要な援助的活動を行うことによって円滑な処理が行われるようにしたり，仕事の連続処理が行なわれるための準備を行う．処理プログラムは，制御プログラムの監視のもとで問題処理の実際的働きをするの

である．

オペレーティング・システムを構成する各々のプログラムは，このように絶えず相互に機能し合いながらその目標達成につとめている．

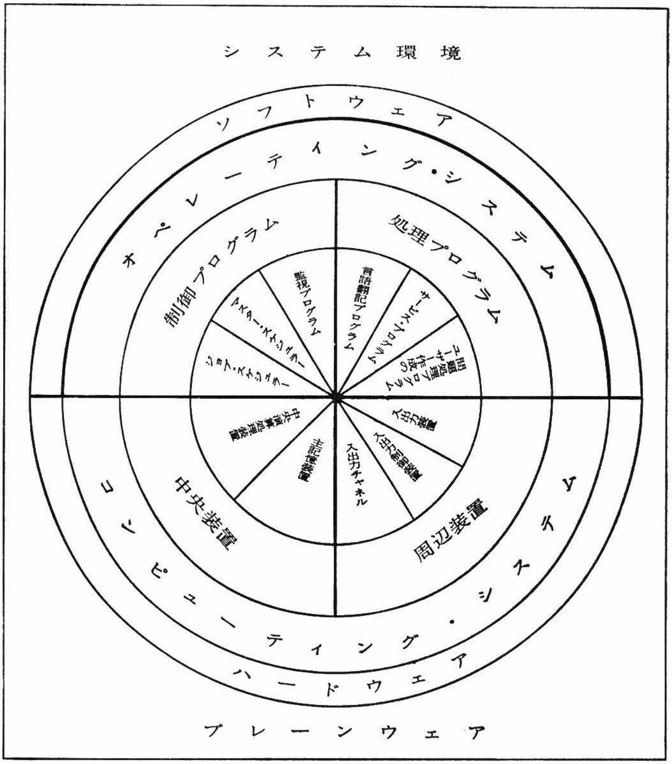
(4) オペレーティング・システムでは，仕事を連続的に処理してゆくためのシステムへの入力単位がジョブである．また，ある仕事が終了したことによる出力単位もジョブである．つまり，オペレーティング・システムを“システム”として把えた場合の入出力単位はジョブという単位である．

(5) オペレーティング・システムは，コンピューティング・システムという環境のもとではじめて機能することができる．一方，コンピューティング・システムをある目的のために作動させているのは，コンピューター部門の管理者，システム分析者，プログラマー，操作員さらには企業全体を構成している人間頭脳である．つまり，オペレーティング・システムは，コンピューティング・システムとそれを活用する人間集団というシステム環境の中でその機能を果しているのである．

オペレーティング・システムを５番目の**システム環境**という観点から把えると**第１図**に示したようになる．

すなわち，情報処理システムという広い領域にわたるシステムとして把握し得るのである．情報処理システムという範ちゅうでコンピューターを把えた場合には，その構成要素は，ハードウエア，ソフトウエアそれにブレーンウエアという三大要素が三位一体をなしており，そのいずれがまずくても，システム全体の最適化はできなくなる．ここでいう**ブレーンウエア**（brainware）とは，コンピューティング・システムやオペレーティング・システムを利用する人間の頭脳にかかわる問題であり，彼らが修得している利用技術や手法だけでなくそれらを身につけさせる教育指導，その利用技法を充分に発揮できるための環境，制度といったすべての無形の知識や技能が包含されている．

先述したようにオペレーティング・システムは，コンピューティング・システムを通じてその機能を果してゆくシステムである．コンピューティング・システムの構成要素である中央演算処理装置，主記憶装置，入出力チャネル，入出力制御装置あるいは入出力装置のすべてを無駄なく働かせ，こき使うように



第1図　情報処理システムの三大構成要素

スケジュールしてゆくのがオペレーティング・システムの**制御プログラム**の役割りであり，その指示にもとづいて問題解決に専念するプログラムが**処理プログラム**である．したがって，オペレーティング・システムは，それ自体で一つのシステムを構成しながらも，コンピューティング・システムを不可欠の相棒としており，それを通じてのみ自己の目標を実現することのできるシステムである．つまり，オペレーティング・システムは，ハードウエアと一体となって情報処理システムの中核を構成しているのである．

一方，オペレーティング・システムは，コンピューターを利用する人間に対しても深いかかわりをもっている．人間に奉仕するために，人間の道具としての意義しかもたないコンピューターの利用技術面に貢献するオペレーティング

・システムであるのだから，このことは至極当然なことである．コンピューティング・システムがいかに高速処理能力があり，オペレーティング・システムがいかに高尚な哲学のもとに設計されていようとも，それらが真価を発揮できるかどうかは，すべて利用者の頭脳にかかっている．とはいうものの，利用者がコンピューターのあまりに細部のことや，余分なことに注意を払わないでその利用法にだけ努力を傾注できるようになっておれば，それだけ効果的利用が期待できよう．このための配慮が充分なされているのがオペレーティング・システムである．

したがって，われわれがオペレーティング・システムに構造的アプローチをするためには，単にオペレーティング・システムの領域内だけに注目するのではなく，ハードウエアやブレーンウエアとの関係でそれぞれの構成要素がどう位置づけられているかにも注目してゆく必要がある．そのためには，情報処理システム全体のかねあいの中でどのような設計目標がオペレーティング・システムにもたされているかを検討してみることこそが重要となる．

二. オペレーティング・システムの設計目標

1. スループットの向上

たとえば，単線型の組立てラインを想定してみよう．かりに，自動車のある部品を組立てるラインと考えてもよい．この自動車部品を組立てるある種の作業過程は比較的単純で1時間100個の処理ができるとしよう．ところが，別の作業は複雑で一時間35個しか部品の取付けが出来ないとする．この場合の一時間の完成部品はいくつになるであろうか．極めて当然のことながら，後者の作業による遅滞が生じ，結果としては単位時間当り35個の完成部品の生産となる．前者の作業過程では実に65%の遊休時間が生じることになるのである．しかも必ずしも何時も35個一時間で生産出来るとは限らない．作業を開始するに当っての準備時間，作業員の交替時間あるいは故障時間も見込まなければならないであろう．いずれにしろ，物事がシステム化されれば，されるだけ全体的

なバランスのうえでその生産性が検討される必要性がでてくる．

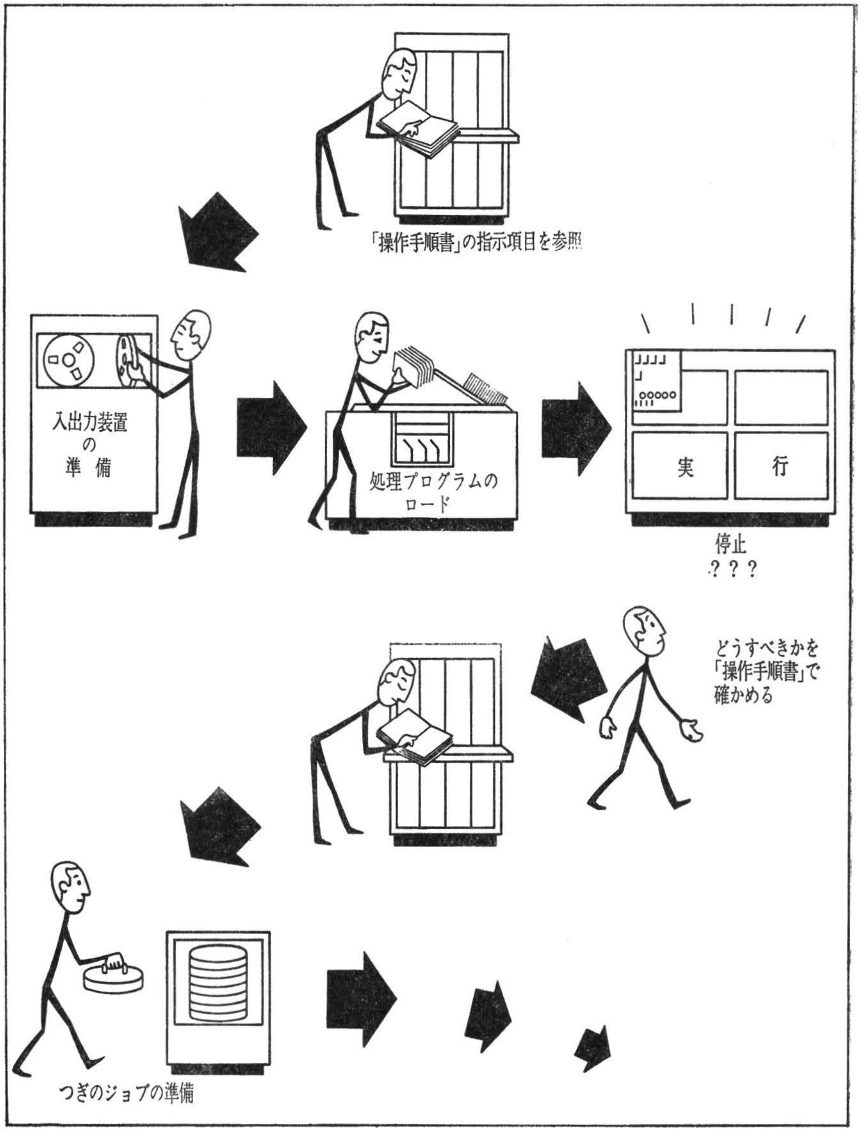
この極めて単純で明快な理屈に関して過去のコンピューター界は誤解をしてきていたのである．もっともこの誤解は現在でも根強いようである．つまり，これまでのコンピューターの生産的能力が中央演算処理装置や入出力装置個々の処理速度を尺度として云々されてきたことである．たとえば，1秒間に1万回の掛算をする能力をもっているとか，カード読取装置が1分間2,000枚の読取りをするとかいった具合いである．

これらの能力が実際に発揮されることは，100回に1回もないかも知れない．ちょうど，自動車の最高時速が180キロ・メートルと表示されていても，実際にそんな速度でぶっとばすことが出来ないのと同じ事態である．ところが，実際にはシステムを構成する個々の装置の性能だけでもってこれまではその生産的能力を評価してきたのである．

コンピューターの内部速度が増すにしたがって，このような評価方法に疑問がもたれるようになり，コンピューターの各所に生じる遊休時間をどのようにして最少限にするかという問題提起となって表面化した．これを解決するという目標のもとにオペレーティング・システムの設計概念が作り上げられたといってもよい．つまり，個々のシステム構成要素の性能を問題とする非システム的アプローチからシステム全体としての生産性の向上ということから，個々の要素の性能を生かしてゆくという極めてシステム的なアプローチへと進んだのである．ここに出てきたのが“スループット”という概念である．

スループット（throughput）とは，システム全体の効率を測定する単位であり，単位時間当りにコンピューティング・システムがどれだけの生産性をあげるかを問題としたものである．オペレーティング・システムの第1の設計目標は，このスループットを最大限に向上させることにある．

では，スループットを向上させるためには一体どのような領域に着目すべきなのであろうか．さきの組立てラインの例でも示唆されているように，システムに表われる遊休時間を最少限にすることによって，個々の装置にもたされている本来の性能に近い能力を発揮させようということになる．他の領域は，機械システムそのものの生産性だけを考慮するのではなく，プログラムの作成や



第2図　非連続ジョブ処理の操作手順

デバッキング等の生産性を高めることによって，その結果としてシステム全体の効率を上げようということである．

そこで，オペレーティング・システムはスループット向上の手段としてつぎのような概念を導入している．

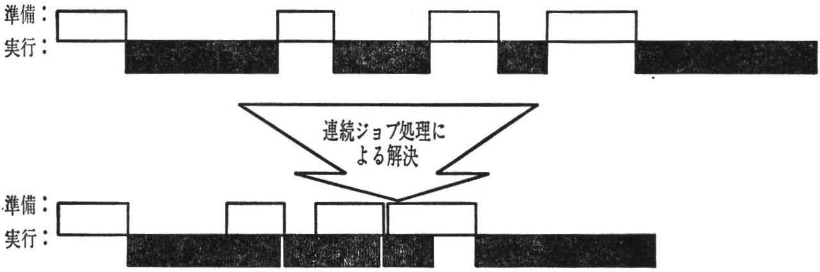
(1)　連続的ジョブ処理

ある仕事をコンピューターで処理させるためには，その仕事の準備のために操作員の介入が必要となる．**第2図**は連続的なジョブ（仕事）の処理が出来ない場合の手順である．最初の仕事を開始するまえに，操作員は「操作手順書」でその仕事を実行するにあたって必要となる行動の内容を知る．この指示にしたがって，その仕事に必要となる入出力データ記録媒体（磁気テープ，カード・デック，印刷用紙など）を所定の入出力装置にセットする．そのあとで，所要のプログラムを主記憶装置にロードする．たちまちのうちに，4，5分は経過してしまうであろうし，またこの時間は，全くコンピューターは休んでいることになる．ともあれ，これらの手続きが終ると，コンピューターはそのプログラムの実行に入る．そのうち，コンピューターは仕事を完全に終了したか，あるいは何らかの誤りか機械の故障のために停止状態になるだろう．操作員は操作卓に引き返して，停止理由を確かめることになる．もし，何らかの誤りによる停止なら，その誤りの原因をつかみ，訂正措置をとる必要がある．仕事が完全に終了したのなら，次の仕事のために上記と同様の手順をとることになろう．

このような非連続処理の場合の準備（セット・アップ）時間とコンピューターの作動時間を対比して図式化したのが**第3図**の上の部分である．コンピューターの実行時間が比較的長いものに関しては，その準備時間は，さほど問題にならないが，処理時間をあまり要しない仕事においては非生産的時間の割り合いは極めて高いものとなる．この問題はコンピューターの性能が向上すれば，ますます深刻化する．

これを解決する手段としては，前の仕事が実行されている間に，操作員が次の仕事の準備をすることができるようにし，かつ，仕事が連続的に処理される

ようにすることである．もし，このような技法が実現できたとしたら，**第 3 図**の下の部分に示したようにコンピューティング・システムの遊休時間を大幅に短縮することが可能となる．



第 3 図　連続ジョブ処理による遊休時間の短縮

このために，オペレーティング・システムにはジョブ・スケジューラーという管理プログラムがある．このプログラムの構想は，操作員の仕事の負担を出来るだけ少なくし，その仕事に先立って，あらかじめ磁気テープや磁気ディスクのような入出力媒体をあいている装置にセットしておけば，自動的にその装置を選択してつぎの仕事にかかるための準備を即時にやりとげるし，もし所要のデータがセットされていないと操作員にその旨の指示をメッセージで行う．もちろん，必要なプログラムをプログラム・ライブラリーから自動的に選択してロードしたうえで実行に入る．ただし，コンピューター使用者は，あらかじめ**ジョブ制御カード**（job control card）と呼ばれるものを作成し，そのカードに実行すべき仕事名，プログラム名，入出力装置の指示をしておく必要がある．

なお，実行過程において誤りとか異例の状態が発生した場合には，マスター・スケジューラーと呼ばれるプログラムがその理由と操作員のとるべき措置を指示してくれる．

このように連続的ジョブ処理という構想のもとに，オペレーティング・システムは，コンピューティング・システムの遊休時間を少なくし，操作員というブレーンウエアに対してはその作業負担を軽減することを狙っている．

(2)　システム資源の最大限活用

連続的ジョブ処理という思想で，コンピューティング・システム全体が一せいに遊休する時間を最少限におさえることが出来るとしても，ある仕事が実行されている間にも個々の装置がなまける可能性は充分ある．つまり，個々の装置は，その本来もっている性能の何分の１しか働かないことになる．たとえば事務計算を例にとると，プログラムの実行中でも中央演算処理装置は１％から30％ぐらいしか働いていないのが普通である．入出力装置に較べてあまりにも処理速度が高いので，このようなアンバランスが生じるのである．

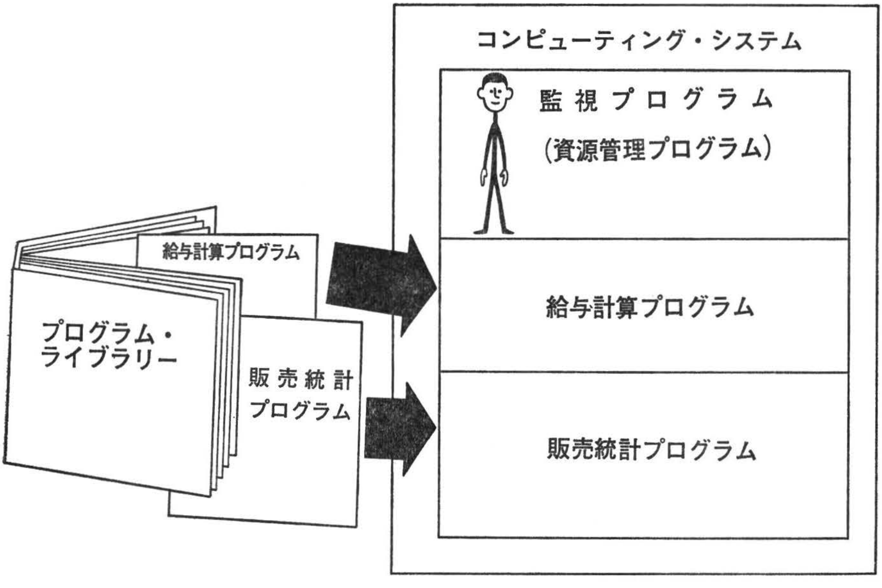
また，ある仕事の実行のために全ての入出力装置が使われることは，まずない．いくつかは遊んでいる．主記憶装置だって，全体が使用されることはあまりない．何割かは，用をなしていない事態がしばしば発生する．

したがって，これらシステムを構成する要素が出来るだけ多忙な状態に保つような方策こそ，スループットの向上のために必要となってくる．このためにシステム資源の最大限活用という使命がオペレーティング・システムの設計目標として課されたわけである．

システム資源（system resourse）とは，仕事（ジョブ）を実行するために必要となるコンピューティング・システムとオペレーティング・システムの全ての機能構成要素を意味する．具体的には，中央演算処理装置，主記憶装置，入出力チャネル，入出力制御装置，入出力装置といったハードウエアと入出力ファイル（普通データ・セットと呼ばれる）および制御プログラムと処理プログラムからなるオペレーティング・システムもシステム資源に変りはない．ただし，ここで特に問題となるのは前者，つまりコンピューティング・システムを構成しているシステム資源である．

オペレーティング・システムに課された任務は，これらシステム資源が最大限活用されるように資源の割り振りを行うことである．したがって，これを達成するためには，利用可能なシステム資源はどれであり，それぞれの仕事で必要とする資源はどれとどれをどの程度といったことを絶えず監視してゆく必要がある．この考えをおし進めてゆくと，終局的には，同時併行的に実行される２つ以上のプログラムにシステム資源をうまく割り振ることはできないものか

ということになる．その結果，２つ以上の全く独立したプログラムを同時併行的に処理してゆくという構想がでてきた．これが，**多重プログラミング**(multi-programming)，**多重処理**(multi-processing)あるいは**多重ジョビング**(multi-jobbing)という考えとなってオペレーティング・システムの重要な設計目標となってきた．つまり，オペレーティング・システムでは，システム資源を同時併行的に使用することによって，一台のコンピューターで一度に複数個の仕事を実行してゆくという多重プログラミング方式でシステム資源を無駄なく最大限に利用しようというわけである．



第4図　多重プログラミングの構想

第４図に図示したように主記憶装置に２つ以上の全く独立したプログラムを呼び込み，それぞれの仕事で必要とする入出力装置を割当てる．中央演算処理装置でのプログラムの実行は，一方のプログラムの実行中に待ちが生じたとき（たとえば，入力装置からデータを読み込むような場合待ちが生じる），他方のプログラムの実行を行う．このような待ちが生じたかどうかとか，システム資

源のどれが現在使用されており，どの部分が遊んでいるかを絶えず監視するのが監視プログラムの**資源管理プログラム**（resource management program）と呼ばれる部分である．

ところで，このような考えを拡大してゆくと，個々のプログラムの実行の優先順位が問題となってくる．どちらのプログラムもすぐに実行可能な（待ちがない）状態にあるとき中央演算処理装置での処理の権利はどちらにまず与えるべきかという場合を想定してみるとよい．何らかの優先順位が与えられていた方が処理しやすいはずである．2つ以上のプログラムがあるシステム資源を競い合っているときには，優先順位の体系で処理するという概念がオペレーティング・システムに導入されることになる．

(3) 使いやすさの向上

連続的ジョブの処理が可能になり，システム資源の最大限利用が実現できたとしても，それによってブレーンウエア，とりわけプログラマーの負担が増大するようなことになったのでは，システム全体のスループットに充分寄与したとは言えまい．

オペレーティング・システムは，あらゆる適用業務に関してスループットの向上に役立たなければならないということから多種類のプログラミング言語を自己の支配下におくことをその目的とする．アセンブラーはもとより，事務計算分野に適したCOBOLやRPG，科学計算分野向きのFORTRANやALGOL，あるいは，事務計算と科学計算との両方の領域の特性を兼ね具えた経営計算の分野を指向したPL/I言語などが使えることが前提になるし，データ通信用プログラムなどが容易に作成できるような言語が用意される必要がある．

一般にプログラムの設計が終了したあとプログラマーの負担が大きくなるのは，そのプログラムのテスト段階におけるデバッキングである．このデバッキングのためにプログラマーの生産性は著るしく低下する場合さえある．したがって，この面での生産性を向上させるようなソフトウエアの工夫，つまり，デバッキングの手段を提供することもオペレーティング・システムの目標となる．さらには，ハードウエアの故障個所を早期に発見し，故障時のダウンの時

間を短縮させることによってその使用時間を向上させることもスループットという観点からすれば重要な要素といえる.

2. 応答時間の短縮

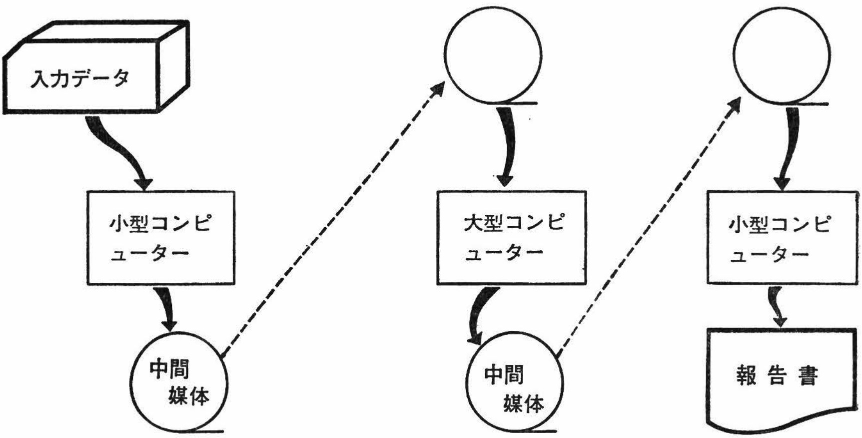
応答時間とは, ある事柄に関してコンピューターの行動が要求されてから, その要求に対する処理がとられてその結果がコンピューターから出力するまでに経過する時間のことである. 一般にこの応答時間のことを**ターン・アラウンド・タイム** (turn-around time) と呼んでいる.

給与計算のようにデータを一まとめにして一括処理を行うような場合には, その応答時間が長くてもさほど問題とならないであろう. 一方, 銀行の窓口とかみどりの窓口における処理のように要求の都度, その場その場で処理しなければならない場合には迅速な応答が要請される. また, 一括処理形式の場合でもスケジュール化されていない仕事 (例えば, 重役から, これこれの資料を何時までに欲しいと要求されるような場合) が飛び込んで来るときには, 多くの場合限られた時間での応答が必要となる.

しかしながら, 一般にバッチ処理方式の場合には, 応答時間は比較的長い. この方式の場合には, 要求元がデータを計算室に渡し, 一定時間が経過したあとに印刷された結果を受け取る. ところが, バッチ処理方式と即時処理 (real time) 方式とを併用しているようなシステムでは, 応答時間はさまざまである. さきの例で示したようにバッチ処理方式では応答時間は長くてもよいが即時処理方式に要求される応答時間は短かいほどよい.

たとえば, ある種の適用業務では, ミリセカンド単位あるいはそれより短かい応答時間を要求する. これに反して, 鍵盤装置から通信回線を通して入力するといった処理の場合には, 数秒から数十秒といった応答時間が望ましい. 種々の適用業務やさまざまな応答時間が要求される環境が想定される以上, それぞれに見合った応答ができるようなシステムを設計することが要求される.

ところが, 初期の大型コンピューターでの処理は**第5図**に示すような方式をとっていた. 大型コンピューターであるだけに内部処理速度ははやい. 多重プログラミング方式を採用していないオペレーティング・システムであるためこ



第5図　初期のオペレーティング・システムにおけるターン・アラウンド

の大型コンピューターの入出力装置としてカード入出力装置や印刷装置のような低速なものを使ったのでは，内部処理と入出力処理に格段のアンバランスが生じシステム資源の利用効率はさがってしまう．苦肉の策として小型コンピューターを衛星コンピューター用に導入し，入力カード・データは一たん磁気テープに移しかえたうえで，大型コンピューターの入力として使用する．一方，出力の方も磁気テープにとったうえで衛星コンピューターで最終報告書を作成する．つまり，一つの仕事を完成するために3回のプログラムの実行を行うことになるだけでなく，再三再四操作員の介入を必要とする．これではターン・アラウンド・タイムは非常に長くなってしまう．それゆえに現在のオペレーティング・システムでは，仕事を最初から最後まで一つのシステムで達成することをその目標としている．さきにふれたようにシステム資源を最大限に活用するために多重プログラミング方式をその基本にしているので，カード上のデータを磁気テープに移すような仕事は，他のジョブと同時併行的に実行できる．それに，過去のシステムのように衛星コンピューターから大型コンピューターにデータを移動させるような操作員の介入も一切不用となるので，その応答時間は向上する結果となる．

瞬時にかつ遠隔地点からの応答にこたえるためには，当然のことながら，中央のコンピューターと遠隔地にある端末機器を通信回線で直結することによって応答時間を短縮することになる．オペレーティング・システムにはデータ通信回線を用いた遠隔地点からの入力が，完全にかつ迅速に処理し得るようなプログラミングの機能と通信回線制御の機能が賦与されていなければならない．

さらに一括処理の場合でも，割込みの特急ジョブに対して充分こたえられなければならない．これに対処するためには，特急，急行，快速，普通といった段階の処理が可能になるような工夫が必要となる．このための設計概念はジョブ毎に優先順位を与えるという方式である．優先順位の高いジョブは，いくつかの入力ジョブのうちから優先的に選び出されて，他のジョブよりも先行して実行されるシステムである．この方式が適用されるジョブ・スケジューラーを**優先式スケジューリング・システム**(priority job scheduling) と呼んでいる．

このようにオペレーティング・システムでは，さまざまな適用業務のさまざまな応答時間に対して最適応答が出来ることが重要な目標となる．

3. ブレーンウエアへの貢献

コンピューター利用に直接関係をもつコンピューター室の管理者，プログラマーあるいは操作員のわずらわしい雑用を最少限にして，思考性を伴った頭脳的で創造的な仕事を行なわせたいという期待は多くの人々がいだいている．ブレーンウエアにやらせるには，あまりに無駄が多く労力を要する仕事をコンピューターに肩がわりさせるための機能をオペレーティング・システムに具備させたいという目標がでてくるのは当然のなりゆきといえよう．このことによって，情報処理システムの全体としての生産性を高めると同時にシステムとしての最適効果が達成し得るからである．

(1) プログラマーの生産性の向上

オペレーティング・システムは，プログラマーが，与えられた問題を解決するためのプログラムや特定の適用業務のプログラムを作成する段階での作業を援助するためにつぎのような諸機能をもっていることが前提となる．

各種各様の問題解決や適用業務のプログラム作成に十分適合し，かつプログラム作成が容易に行いうるような言語翻訳プログラムが備わっていること．たとえば，科学技術計算のプログラム言語としては FORTRAN や ALGOL が，事務計算のものとしては COBOL や RPG が，あるいは両用のためや経営計算の目的として PL/I が利用可能になっていることが必要となる．

適用業務が高度化してくると，必然的に作成すべきプログラムは複雑化し大きくなってしまう．したがって，複数人のプログラマーが1つのプログラムの完成のために従事せざるを得なくなる．そのためにはプログラムの分割作成を余儀なくされる．分割して作成したプログラムは，そのプログラムを実行する以前に完全な一つのプログラムとして結合されなければならない．このようなプログラムの分割作成が容易に行え，かつそれぞれの分割プログラムの連係結合を達成する機能も要求される．

プログラムの分割作業が可能になれば，単にプログラマーの分担作業によって短期間にプログラミングが実現できるという利点があるだけでなく，プログラムの設計段階での作業も容易になるし，プログラムのテストも各部分のテストを完全に行ったあとで全体を結びつけることができるので，プログラム・テストの効率化が計れる．

使用頻度の高いルーチンやプログラムを必要のつど作成するということは，プログラマーの生産性を著るしく低下させてしまう．一度作成したルーチンやサブプログラムが各種適用業務で容易に共用できる体制が完備していることが望ましい．そのためには，既成ルーチンやプログラムをライブラリーに格納しておき，必要のつど容易にしかも即時にとりだして利用できるような機能が要求される．しかも，原始プログラムの状態，目的プログラムの状態，実行可能なプログラムの状態といったあらゆる形態で格納し得るライブラリーが必要であり，かつ，あらゆる形態でライブラリーにあるルーチンを結合できることが望ましい．

プログラマーにとっての作業内容で大きな負担になるのは，プログラムの正確性とあらゆる条件の網羅性をテストしデバックする段階である．それゆえに，プログラム・テストを容易に行い，誤りの個所の修正や不足条件の追加が

簡単にできる機能も必要となる．

さらには，プログラマーがハードウェアや制御プログラムといったものにほとんど関心を払うことなく，ただ与えられた問題だけに全思考力を集中してプログラムの作成が行いうるようにオペレーティング・システムが貢献することも要求される．

(2) 導入管理者の補佐機能

コンピューター室の管理者が管理者本来の意志決定的仕事に従事できるようにするためには，仕事のスケジューリングとかデータやプログラムの管理といったわずらわしい仕事からできるだけ解放される必要がある．したがって，このような仕事をオペレーティング・システムの援助によって，できるだけコンピューティング・システムのもとで処理することである．このため具体的機能としては，つぎの諸点が挙げられる．

- ●コンピューティング・システム（とりわけ，磁気ディスクやドラムのような外部記憶装置）に貯えられているデータやプログラムの自動的管理と使用が可能になる．
- ●処理すべき作業（ジョブ）に優先順位を示すコードを与えることによって，優先順位の高いジョブから順番に処理してゆく．
- ●機械室の運営管理上必要となってくる情報（たとえば，ジョブ毎の処理時間とかカードや用紙の使用枚数あるいは機械の使用料金）を容易に取り出すことができる．

(3) 操作員への手助け

オペレーティング・システムがジョブを連続的に処理してゆくことによって，システムのスループットの向上を計ることを目標にしている以上，操作員のたび重なる介入によってシステムの作動がストップすることは許されない．したがって，操作員への手助けの主眼は操作員の余分な判断を最少限にとどめ，操作自体を簡素化することにある．

そのためには，ジョブの処理過程においてのシステムと操作員とのやりとり

は簡潔で明解な指示で行う必要がある．システムから操作員への依頼事項はメッセージの形式で印刷されるか表示装置上に示されるべきであるし，操作員がシステムに何らかの指示をするときには，簡単な指令語を打鍵するか，表示装置上にライト・ペンで指示するとかいった方法である．

新しいジョブを処理するために，必要となるデータとかプログラムを操作員が判断して準備をするとなると，必然的にシステムの中断が生じてしまう．これを避けるためには，オペレーティング・システムの制御プログラムが各ジョブに必要となるデータ群とプログラムの情報を入手し，自動管理をすることである．そして，必要となるデータ・ファイルをどの装置にセットすべきかの指示をあらかじめ操作員に与えることによって中断を最少限にすることが望まれる．

さらには，プログラムの実行過程において発生したハードウェアの異常事態を自動的に記録にとどめておき，ハードウェアの故障の発見に役立てることによって，それによる中断を最少時間にくいとめるような機能も必要である．

以上のように，システムを運用してゆく人々に対しても最大限の考慮が払われて設計されているのがオペレーティング・システムなのである．

4.　システムの適応性と開放式設計

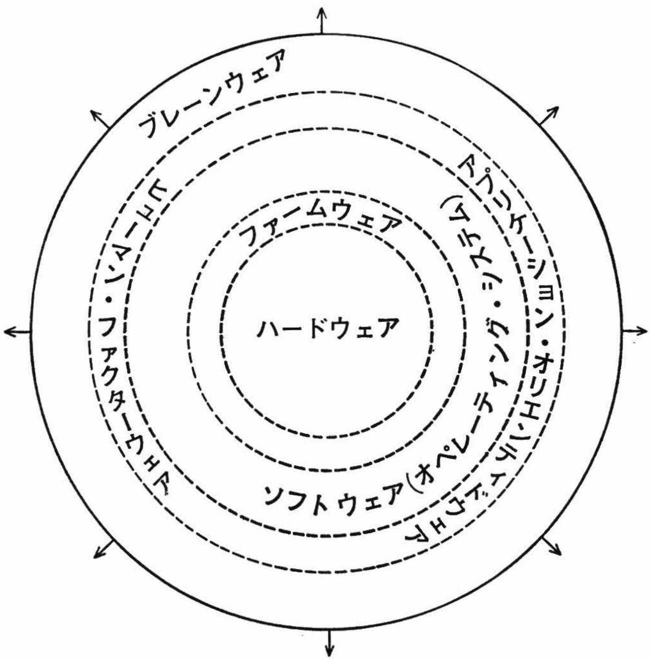
コンピューターの適用分野はめざましく拡大してきた．初期のコンピューターは事務計算向きか科学計算向きかで区別されていたが，今や一つのコンピューターであらゆる適用分野をカバーするようになってきた．事務計算や科学計算はいうまでもなく，データ通信やプロセス制御あるいは学習制御といったさまざまな分野に同一コンピューターが導入される．しかも，コンピューティング・システムそのものも，適用業務と用途に応じて無限の機械の組合わせができる．つまり，コンピューターで処理する適用業務の種類が拡大化していると同時に，機械構成の方もそれぞれの特徴を生かした構成をとるのである．そのうえ，一たん導入されたシステムでもその利用方法と適用業務の拡大成長につれて，その機械構成自体が変化してゆくことが予想される．したがって，コンピューティング・システムと適用業務との仲立ちの役割りを果すことを目的と

したオペレーティング・システムは変化と多様性に適応するように設計される必要がある．そのために，オペレーティング・システムが多種多様な適用業務に見合ったプログラム機能を備えていることはもとより，機械構成とその用途に応じてオペレーティング・システムの機能を幾通りにでも組合せることができるような考慮がいる．多彩な組合せを実現するためには，変化に応じることのできる**モジュール概念**（modular concepts）が重要な役割りを果す．モジュールとは，単位という意味である．それぞれの機能ごとにある単位にまとめてしまう．つまり，ある機能部分に対して一つの完結したサブルーチンにしてしまい，サブルーチン間の相互連絡や制御の方法を共通化してしまう．こうすることによって，必要となるいくつかの機能に関するモジュールを容易に組合わせることができるようになるし，追加の機能モジュールでも即座に付加できる．ちょうど，積木を積重ねると同じ方式でモジュールを組合わせることが可能となる．積木の四角形だとか三角形という形がモジュール間の相互連絡や制御方法の共通化に対応する．実際には，プログラム・モジュールの中に一定形式の制御ブロックと呼ばれるテーブルを設けることによって，そのブロックの中に貯えた情報で相互連絡を可能にするし，モジュール間の制御権の移譲を決められた命令によって行うのである．

また，問題処理プログラムの適応性を高める方法として**入出力装置に依存しない**（device independent）プログラムの作成が要請される．“入出力装置に依存しない”プログラムとは，実際に使用する入出力装置の構成には無関係であるようなプログラムを作成し（具体的には，プログラム作成段階では，いっさい特定の入出力装置を指定することはせず単に記号で入出力装置を指定する），プログラムの実行段階で制御プログラムの働きによって入出力装置を割り当てさせるやり方である．このような“入出力に依存しない”方法によって，入出力装置の変更のために生じるプログラムの書き換えを一切不要にする．

情報処理システムは，めまぐるしく変化してゆく．ハードウェアは絶えざる技術革新によってその内部回路が変化するだけでなく，次から次へと新しい入出力装置が輩出する．ハードウェアの利用手法や適用手法もブレーンウェアの働きに応じて変化してゆく．利用分野が拡大するにつれてブレーンウェアの層

は拡充されざるを得ない．それらの変化が相乗されて情報処理システム全体の変化として表われてくる結果となる．



第6図　システムの拡充化傾向

第6図に示したようにハードウェアはソフトウェアの機能を自己の一部とする方向での変化を示すし，ソフトウェアはブレーンウェアの肩がわりをすることを指向してゆく．ブレーンウェアは，ますます外へと拡大してゆく．それぞれのシステム要素が開放的成長をめざしてゆくのである．ソフトウェア機能のハードウェア化は，近時**ファームウェア**という言葉で表現されているものであるが，オペレーティング・システムのある種の特定機能，たとえば制御プログラムの使用頻度の高い定型的ルーチンの機能を内部回路でおきかえたり，言語翻訳プログラム（コンパイラー）のある種の機能ルーチンを**読出し専用記憶装置**(read only storage)に記憶させるといったやり方である．ファームウェアの目的はシステム費用の効率を上げ，システム資源の最大限有効利用をめざし

たものであり，ハードウェアとソフトウェアを有機的に結合しようとする．ともあれ，このような方法によって，ハードウェアとソフトウェアの明確な境界線がなくなる方向へとハードウェアは拡大してゆく．

一方，ソフトウェアは，システムを使用する人間の利用のしやすさという人間工学的要素，つまりヒューマン・ファクターを採り入れる割合を増してゆくばかりでなく，新たな適用業務に見合ったソフトウェアの機能を付加してゆく．

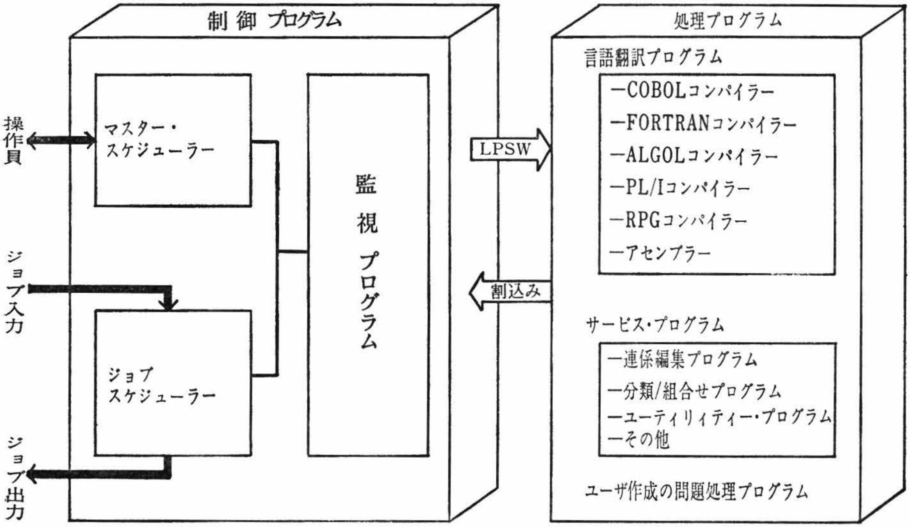
このような変化に耐えぬいてゆくためのオペレーティング・システムは，新しいハードウェアや利用技術を簡単に受け入れて成長してゆくことのできる設計がなされていなければならない．オペレーティング・システムそのものが陳腐化し，時代遅れとなって使いものにならなくなるのではなく，自らも機能上の諸能力を拡張し変化を容易に採り入れることのできる開放式設計であることが必須である．

開放式設計がなされているかぎり，オペレーティング・システムには断絶はまず生じないであろう．第2世代のコンピューターから第3世代のコンピューターには大きな断絶があったが，オペレーティング・システムにはそのような事態は期待できない．第4世代のコンピューターでも第5世代のコンピューターでもその構想が生きつづけてゆくと思われる．

三．システムの構成要素と相互関連性

オペレーティング・システムを使用者の立場からみた場合，その構成は，いくつかの言語翻訳プログラム群と一連のサービス・プログラムおよびこれらプログラムの監督・制御の機能をつかさどる制御プログラム群から成り立っている．さらに，システム全体の管理という観点からすれば，使用者自身が作成した適用業務プログラム自身も，オペレーティング・システムを構成する一部分となり，制御プログラムの監視下におかれることになる．

このことを構造化して示したのが**第7図**である．いかなる条件のもとでも，言語翻訳プログラム，サービス・プログラムおよび使用者作成の問題処理プロ



第7図　オペレーティング・システムの構成要素構造

グラムといった処理プログラムは，制御プログラムの管理下で実行される．個々の処理プログラムが制御プログラムと無関係に実行され，何らかの結果を出力することはあり得ないのである．たとえば，処理プログラムの実行過程において，カードからデータを主記憶装置に読み込みたいといった事態を想定してみよう．われわれの作成するプログラムの中には，確かに READ とか GET とかいった入力命令を所要な個所に記入する．ところが，これらの命令が実際の入力機能を果すのではない．入力の諸準備をするだけで，読取り作業は監視プログラムが行う．つまり，処理プログラムの入出力の要請が出されると**監視プログラム呼び出し命令（SVC 命令）**によって，**割込み**（interruption）状態が生じる．目下実行中の処理プログラムの進行を中断して，制御権が監視プログラムに移譲されてしまう状態である．制御権が監視プログラムに移ると，監視プログラムの中の**入出力監視ルーチン**（input/output supervisor）が実行され，カードの読取りの指示がされる．そのあと，監視プログラムでは「LPSW 命令」を実行することによって，制御権を処理プログラムの割込みが生じた次の命令にかえすことになる．

割込みと LPSW に関しては，次章で詳細にふれるが，ここでは処理プログラムと制御プログラム間での制御権の移動が，「割込み」と「LPSW 命令」によって達成されることに留意していただきたい．処理プログラムの実行過程において，特別な機能の実行が必要になったり，特別な事態が生じた場合には，割込みというハードウェアの機能によって，その処理が監視プログラムに委ねられる．監視プログラムが適切な処理をしたあとで，LPSW という命令を実行することによって，処理プログラムの実行中断点の次の命令から処理が再開される．見かけ上では，全く処理プログラムでの中断が生じなかったように見えるが，実際には非常に頻繁に処理プログラムと監視プログラム間でのやりとりが生じているのである．

このように，処理プログラムは制御プログラムに強く依存しているのであり，それが独立して機能を果すことは不可能である．

1. 制御プログラムの構成と機能

第7図で図示したように，制御プログラムを大まかに把えると，マスター・スケジューラー，ジョブ・スケジューラーおよび監視プログラムの3つの部分から成り立っている．中核となる**制御機能**は監視プログラムの中にある．

監視プログラムは，主記憶装置の割当て，入出力処理の監視機能，多重プロミラミングの同時的実行の制御などの機能を受持つ．マスター・スケジューラーは，操作員とシステム間の連絡という働きを取扱うプログラムである．一方，ジョブ・スケジューラーは，主として入力ジョブの流れ (input job stream) を読取って分析し，入出力装置の割当てなどデータ処理に先立つ準備機能を果すと同時に，ジョブの実行開始と終了後の後始末の働きをするプログラムである．

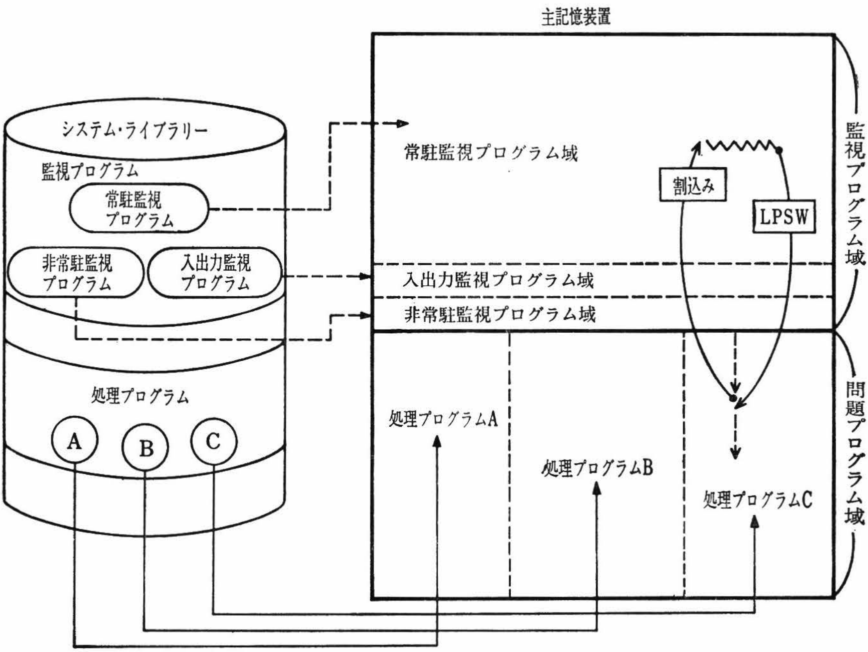
(1) 監視プログラムの機能

第8図で示しているように，監視プログラムは，常駐監視プログラム部分，入出力監視プログラム部分，非常駐監視プログラム部分に大別される．**常駐監視プログラム部分**は，使用頻度が高い監視プログラム・ルーチンの集まりであり，常時主記憶装置に記憶されていて，処理の進行を制御してゆく部分であ

る．“常時”記憶されているという意味は，何らかの仕事が続行されているかぎり，記憶されているという意味であって，物理的に何時でも記憶されているわけではない．つまり，一連の連続したジョブを開始するにあたって，初期手順をとることによって，システム・ライブラリーからローディングされ，つぎの初期手順がとられるまで存在しつづける．

入出力監視プログラムは，その名前の通り入出力オペレーションの開始と監視を行うルーチンの総称であり，その大半のルーチンは，必要のつどシステム・ライブラリーから呼びこまれ，所要の機能を実行する．頻繁に入出力が要請される可能性のあるルーチンだけが常駐ルーチンとなっている．

監視プログラム・ルーチンであっても，たまにしか生じない例外的な事態（たとえば，ある種のエラー）を処理するルーチンは，そのような事態が発生したときだけ非常駐プログラム域にオーバレイの形式で呼び込まれる．これが



第8図 監視プログラムの構成と働き

非常駐監視プログラムである．

この常駐・非常駐の構想は，主記憶装置の効果的利用とシステム全体の処理効率との兼ね合いの問題から生じたものである．使用頻度の高いルーチンを，要請のつどシステム・ライブラリーからローディングしてくることは，時間効率をそこねてしまうので望ましいことではない．一方，まれにしか必要としないルーチンのために常時主記憶装置を確保しておくことは，主記憶装置に無駄が生じることになる．この両面を天秤にかけたうえで，常駐にするか，非常駐にするかが決定される．もっとも，一般的には非常駐的性格をもつルーチンであっても，あるユーザにとっては常駐にしたほうが効率的である場合も想定されるので，各々のルーチンを常駐にすべきか，非常駐にすべきかの最終選択をユーザに委ねることのできるような融通性がもたされたシステムが望ましい．現在でも，この種の選択が可能なシステムがかなりある．

監視プログラムは，さきにふれたようにオペレーティング・システムの中枢制御部分である．処理プログラムの働きを調整し，制御しながら，システムを構成する諸装置の機能とプログラム上の機能とを統一し，効率的な処理が続行していくように制御してゆくことが，その主目的である．監視プログラムは，システムで実行中の複数個のプログラムが互い同志で妨害したり，他のプログラムを破壊しないように調整し，あるいは，制御プログラム自身が処理プログラムからキズつけられないように監視する．ただし，この機能は，監視プログラムだけで達成される機能でなく，記憶保護命令とか入出力命令が監視プログラムのもとでだけ実行されるというハードウェア上の機能とを合体することによって実現されるのである．

また，すでに述べたように，監視プログラムには割込みという手段で，プログラム実行の制御が移譲され，所要の監視プログラムの機能を実行してゆく．この割込みには，制御プログラムの他の部分あるいは処理プログラムから，ある種の処理を行ってくれるようにという要請の命令（監視プログラム呼び出し：ＳＶＣ命令）がなされた結果おこる場合と，入出力操作の完了やプログラムの誤りとか機械の故障に起因して自動的に起る割込みがある．この後者の割込みは，監視プログラムにシステムの諸装置とプログラム上の機能に対して標準

化された制御を行わせるためのものであり，それによってシステム全体の効率化を計ろうとする．

では一体，監視プログラムが果す具体的機能は何であろうか．主要な機能について，まとめておくことにしよう．

1）主記憶装置へのプログラムの読み込み：ある種の仕事を処理するためには，そのための処理プログラムが主記憶装置に呼び込まれなくてはならない．また，プログラムの実行過程において，特別なルーチンを必要とするときにも，それを呼び込む必要がある．このようなプログラムとかルーチンを必要のつど，所定のライブラリーから主記憶装置にローディングする．

2）主記憶装置の割当て：プログラムの実行中に追加の記憶装置を要求する事態がしばしば発生する．実行過程において，特別なサブプログラムを要求することは，そのプログラムが収容できる大きさの主記憶装置を要求することに通じる．また，入出力のための記憶域を要求することもあるし，索引用のテーブルを収容する区域を要求するときだってある．最初から，これらの記憶域を確保しておけばいいじゃないかという議論もでるであろうが，それでは，主記憶装置の利用効率が落ちるし，最初から予測できない場合も多い．また，ある特別な条件を処理するために，まれに必要となる記憶域も多い．

これが，多重プロミラミング処理の場合は，ますます重要性を帯びてくる．ある種のルーチンは，2つの仕事で共有する事態も考えられるし，同一のデータを2つのプログラムでアクセスすることもあり得る．それぞれの記憶域がどのように使われ，現在どのような状態になっているか（共有中か，1つの仕事で使っているのか，遊休中かあるいは全く使っていないのか）を絶えず監視し，主記憶装置の要求のつど，全くさしさわりのない記憶域を当ててゆく必要がある．

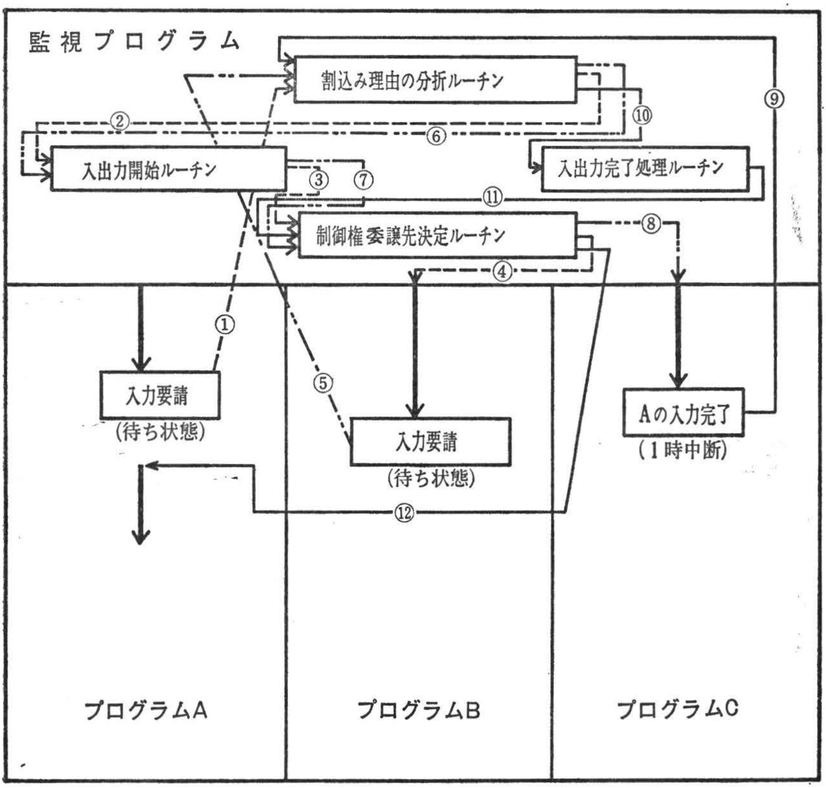
このようなダイナミックな状況下で記憶装置の利用割当てをするのも監視プログラムである．

3）多重プログラムの制御：第8図でも示しているように1つのシステムで同時併行的に複数個の仕事を処理するためのいくつかのプログラムが，主記憶装置に共存し，システム資源を競合し合うことによって，スループットの向

上をはかることがオペレーティング・システムの主要な目標の１つであった．

システム資源の競合のうちで重要なのが，中央演算処理装置（ＣＰＵ）の占有権の問題である．ＣＰＵ利用の制御の流れを図示したのが**第9図**である．この例では，プログラムＡがＣＰＵを利用する優先権が一番高く，そのつぎがプログラムＢ，一番低いのがＣであると仮定した．（プログラムに優先順位があるという考え方は正しくないが，現時点では説明の都合上そうした）．

そうすると，まずプログラムＡがＣＰＵを占有してその実行を開始する．



第9図　多重プログラムの制御

ところが実行過程である種の入力の要請が起ると割込みが生じて，監視プログラムにCPUの制御権が移動する（①）.監視プログラムで割込みの理由が分析され，入力要請であることが判ると**入出力開始ルーチン**が実行される（②）.入出力ルーチンで入力の開始が始まると，CPUの制御権を処理プログラムのどこにもどすべきかを決定するために**制御権委譲先決定ルーチン**が実行される（③）.プログラムAは，目下要請している入力処理が完了するまでつぎの命令を実行することが出来ないので待ちの状態にある．プログラムBとCは，待ちの状態にはないのでともにCPUを利用できるのであるが，プログラムBの方が優先権が高いので，Bの実行が開始される（④）.

プログラムBにおいても，実行過程の途中で入力要請があったとすると，やはり割込みが起り，監視プログラムに実行権が委譲され，プログラムAの場合の入力要請と同じ経路をたどることになる（⑤～⑦）.制御権委譲決定ルーチンでは，プログラムAとBがいずれも入力完了を待っている状態にあることを確認することによって，プログラムCの実行を開始する（⑧）.

プログラムCを実行している過程で，プログラムAの入力が完了したとすると自動的に割込みが生じて，監視プログラムの**割込み理由分析ルーチン**に制御権が移る（⑨）.その結果，入力処理が完了したことが判明するので，**入出力完了処理ルーチン**に移り，正しく入力操作が完了したことを確かめたうえで制御権委譲先決定ルーチンに進む（⑩～⑪）.プログラムAの入力待ち状態は完了したので，プログラムAの中断点から実行が再開される（⑫）.

このように，割込み処理の完了後，待ちの状態の有無と優先順位に応じて，プログラムの実行を決定してゆく．

監視プログラムは，同時にいくつかのプログラムを制御しながら，多重プログラミングを実現し，システムのスループットの向上に機能しているのである．

4）入出力オペレーションの開始と監視：これまでにみてきたように入出力装置からの物理的データの読み込みや書き出しは，監視プログラムの機能である．入出力の開始作業とそれが正しく完了したかどうか，エラーが生じた場合の処理など全ての責任を監視プログラムが果す．

入出力オペレーションを監視プログラムが行うことの理由は，単にそうすることによって，入出力プログラミングという繁雑な仕事からプログラマーを解放することだけにあるのではなく，多重プログラミングのような状況下にあって，入出力オペレーションを効率的にスケジュールし，入出力の待ち時間におけるＣＰＵの遊休時間を他のプログラムにふり向けることによって，システム全体の効率を上げるために必須なこととなるのである．

5）時間の記録に関する機能：ある種の仕事においては，一定の制限時間にもとづいて仕事を打切ったり，報告書のデータ項目の一部に時刻を入れたりしなければならない．また，機械室を管理する立場からすれば，個々の仕事の使用時間を正確に把握し，会計コントロールの情報としなければならない．

監視プログラムは，個々の仕事に要した時間とか，制限時間あるいは一日の時刻に関する管理を行い，その要約的情報を提供してくれる．

6）例外的状態の監視：プログラムの実行中に生じるさまざまな例外的事項，たとえば，除算においてゼロで割ってしまうとか，演算における桁あふれ，あるいは，多重プログラミングにおいてあるプログラムが別のプログラム部分を破壊しかかったような例外状態を監視し，それぞれに対して標準的な措置を行う．

7）エラーの記録：ハードウェアが故障し，その修理に時間がとられてしまうと，結果的にはシステムの生産性は著るしく低下してしまう．まずハードウェアそのものの信頼性が高いことが先決であるが，つぎには，不安定な個所が未然に発見され，手が加えられることであり，故障した場合でもその原因が早期に発見され短時間で修復されることである．そのために監視プログラムは，各入出力装置，制御装置あるいはチャネルといったシステム構成装置のそれぞれで生じたエラーの分析データを全て外部装置に記録する役割りを果す．エンジニアは，定期診断にこの記録を参考にして故障を未然に防ぐことができるし，万が一故障が生じたとしてもその原因を容易に追跡することができる．

監視プログラムに最低限要求される機能は以上のようなものである．視監プラグラムの果す機能の目的は終局的にはシステム全体の効率を上げること

である.

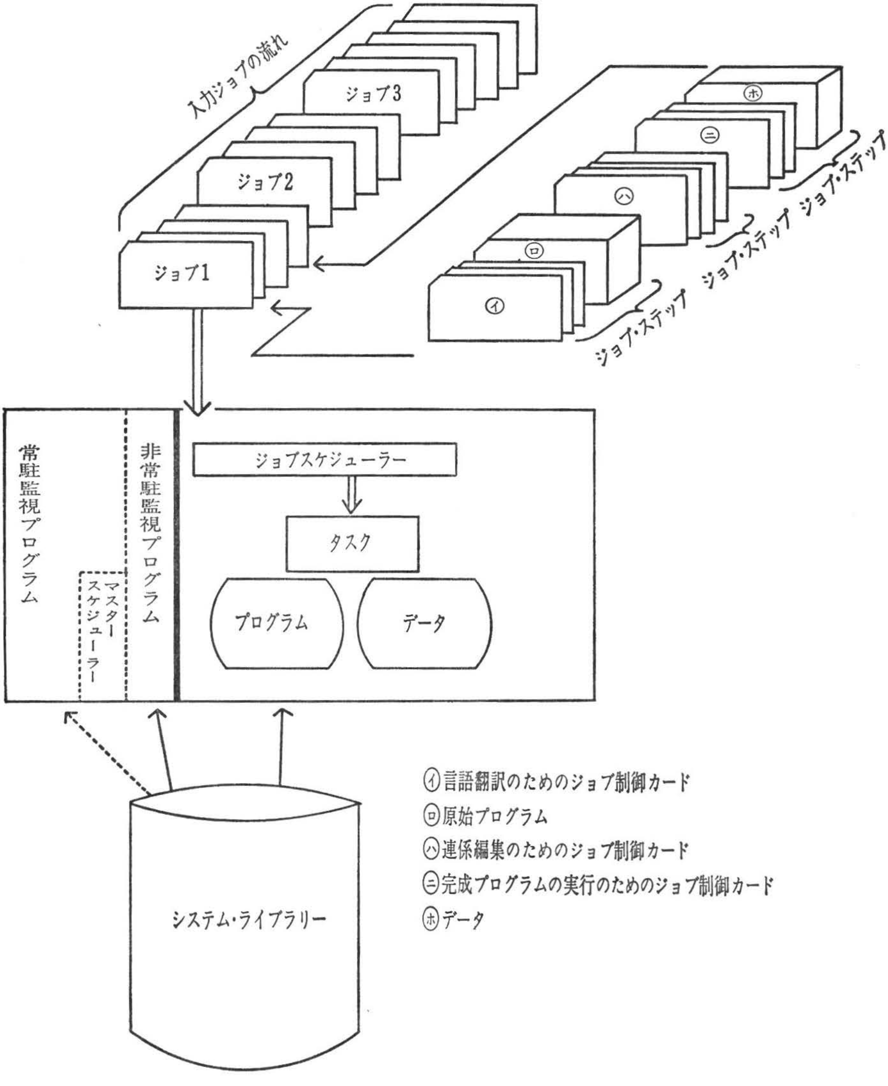
(2)　ジョブ，ジョブ・ステップ，タスクの関連

導入したシステムを利用する場合の会計事務処理上あるいは機械室のオペレーションの立場から見た場合の仕事の基本的単位は，ジョブである．ジョブの根本的性格は，他のジョブから独立しているということである．つまり，他のジョブとの一切の依存関係はないことを原則とする．あるジョブの影響によって，別のジョブの実行が途中で打切られることはないし，あるジョブの出力が他のジョブの入力となったりはしない.

あるジョブに要求される諸条件，すなわち，使用するプログラムだとか，入出力データ・ファイルの属性などは，**ジョブ制御カード**で指定する．ジョブに関する諸条件が指定されたジョブ制御カードは，1つにまとめられて**入力ジョブの流れ**を構成する．入力ジョブの流れには，入力データを含めることもできるが，その主要な目的は，各ジョブを定義し，各ジョブに要求される諸条件を明らかにすることにある.

第10図に示しているように，入力ジョブの流れは，処理を必要とする一連のジョブの制御カードで構成する．システムの利用者は，適切なジョブ制御カードを入れることによって，1つのジョブを幾つかの**ジョブ・ステップ**(job step)に分割することができる．1つのジョブを構成する各ジョブ・ステップは，お互に直接または間接的な関連をもったものである．あるジョブ・ステップの出力が，後続するジョブ・ステップの入力となる場合もあるし，同一の入力ファイルが，2つのジョブ・ステップで利用される場合もある．前者の例としては，最初のステップで実行した結果の出力が次のステップの分類作業の入力となり，分類した結果が第3のステップの報告書作成のための入力になるといった具合いである．後者の例としては，取引ファイルを入力として第1のステップでは在庫更新の仕事を行い，第2のステップでは，請求書作成に使うという場合である.

間接的にしか関係をもたないジョブ・ステップとしては，次のようなものが考えられる．たとえば，3つのジョブ・ステップがあって，第一のステップでは，アセンブラー言語で書かれている原始プログラムをアセンブラーで翻訳し



第10図 ジョブ，ジョブ・ステップ，タスク

第二のステップでは，COBOLで書かれた原始プログラムをCOBOLコンパイラーで翻訳し，第3のステップでは，第1と第2のステップの出力である目的プログラムを連係編集プログラムを使って結合して，一つの完成したプログラムを作る，この場合の第1のステップと第2のステップは直接の関係はない.

第10図の右上に示しているジョブ・ステップは，3つからなっており，第1のステップで原始プログラムを翻訳し，第2のステップでは第1のステップの出力である目的プログラムを連係編集プログラムによって実行可能なロード・モジュールに変え、第3のステップでそのプログラムを使ってデータ処理を行う場合を想定したものである．ジョブ・ステップは，相互に関連をもっているので，前のステップが正しく完了したかどうかを後続するステップに影響させることもできる．たとえば，第1のステップの実行が正しく完了しなかった場合には，第2のステップの実行をスキップして，第3のステップから実行させるという方法が，ジョブ制御カードの指定で実現できる．また，あるジョブ・ステップの途中で実行が打切られると，後続のジョブ・ステップの実行は中止されてしまって，完全につぎのジョブに移ってしまう.

ジョブ・ステップの別の特性としては，あるジョブを構成するジョブ・ステップは，同時点では1つのステップしか処理されないことにある．ジョブの場合は，各ジョブが独立しているので，複数個のジョブ内のステップを併行して実行することは可能であるが，同一ジョブ内では許されない．したがって，ジョブを構成するステップは，処理目的にしたがって，あらかじめ意図した順序に並べておく必要がある.

再び第10図にもどろう．第10図に示しているようにジョブの諸要件を指示したジョブ制御カードは，ジョブ・ステップ毎にジョブ・スケジューラーへの入力データとなる．このジョブ制御カード上に指示されている情報にしたがってジョブ・スケジューラーはプログラム実行のための準備をする．ジョブ・スケジューラーが実行の諸準備を終えると，そのジョブ・ステップは**タスク**（task）として指名される．タスクは，該当のジョブ・ステップで指示されたプログラムのもとで実行される仕事を意味する．タスクは，その仕事を実行するプログラムとデータで構成されるといえる.

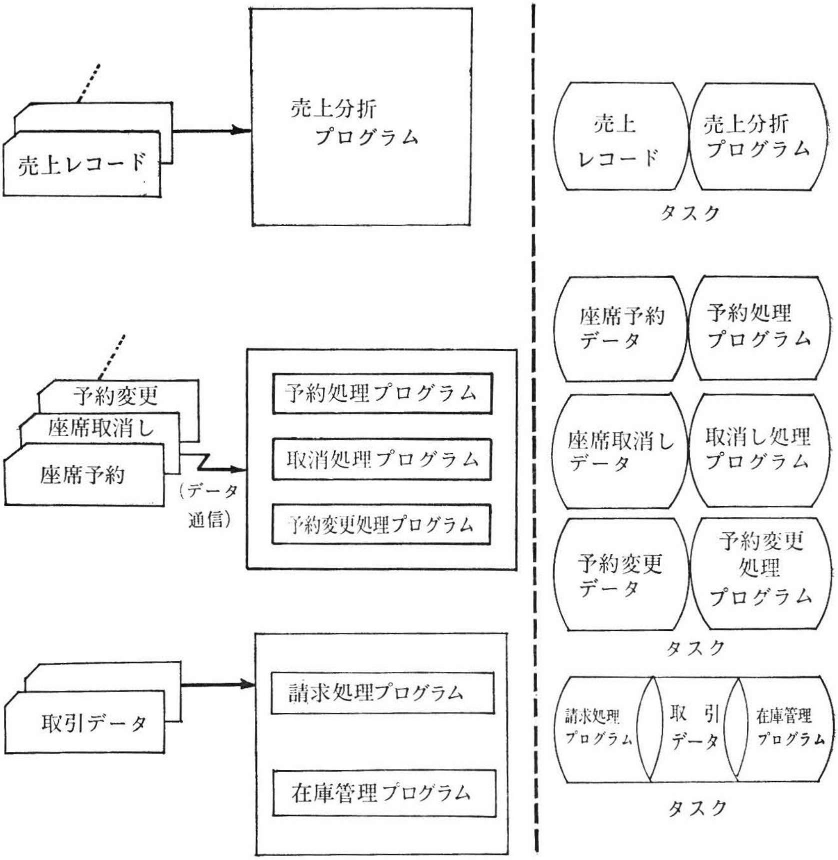
システムを使用する人間の立場からみた場合の仕事の単位はジョブ（またはジョブ・ステップ）であるが，システム内，とりわけ監視プログラムの制御単位は，ジョブとかジョブ・ステップではなく，完全にタスクを単位とした制御なのである．

では何故に，「タスク」というやっかいな概念をオペレーティング・システムに導入せざるを得なかったのか考察してみよう．

初期のオペレーティング・システムでは，一連のプログラムの流れにしたがって，順番にプログラムを実行し，各プログラムとそれぞれに関連する入力データとで１つの仕事あるいは処理すべき問題を構成していると考えるだけで十分であった．つまり，ジョブ（またはジョブ・ステップ）が，システム内での制御単位にそのまま通用した．

ところが，即時処理システムであるデータ通信システムが出現するにおよんで，このような考えでの制御の仕方には限界が生じた．つまり，端末装置から投入されるデータは各種各様であり，多数の処理プログラムの中の特定の１つのプログラムと関係してくる．したがって，入力データと特定プログラムが必ず結びつくという関係は存在しなくなる．**第11図**の中央に示したように入力されるメッセージ（データ）のタイプが異ると，それぞれに対応する処理プログラムで処理されなければならない．しかも，座席予約だけの要求を幾つかまとめて処理することは，即時処理という性格から許されないことなので，各々の要求が発生するつど処理しなければならない．それゆえ，前に入ってきたメッセージを処理し終る前に別の種類の新しいメッセージが多数システムに入ってくることさえ想定される．このような事情のもとでは，ジョブ（または，ジョブ・ステップ）を単位として，システム内で制御してゆくことは不可能となってしまう．

その結果，種類の異るメッセージは，それぞれ独立したタスクとして処理されるという概念が導入されることになる．データ通信システムに要求されるもう一つのものは，同時に併行して多数のタスクが処理できる能力であり，あるタスクに入出力の待ちが生じてＣＰＵを使用できない場合には，別のタスクがＣＰＵを使うことができるような制御方法が必要となる．



第11図　タスク概念の構造

ここにおいて，オペレーティング・システムの設計者たちは，任意の時点に多数のタスクがシステムの資源を競合しながら使用することができるオペレーションの方式，つまり，**多重タスク・オペレーション** (multi-task operation) の概念を導入したのである．つまり，これまでふれてきた多重プログラミングは，多重タスク・オペレーションの一種である．

多重タスクの概念が導入されてくると，第11図の下に示したように，一つのデータが，同時点で2つの異った プログラムで処理される事態さえ予想される

る．しかし，タスクがプログラムとデータの組合わせからなると仮定すれば，2つのタスクに同一データが共有されているに過ぎないということになり，その制御に混乱は生じない．

タスクという概念は，あらゆるジョブの処理形態をシステム内で制御し得るための重要な単位であり，かつシステム資源を適切に割りふってゆくための単位を構成するものである．さきにふれた監視プログラムの機能の多くは，このタスクを単位とした管理を行う目的のためにあるので，一般にこれらのプログラム群を**タスク管理プログラム**（task management program）と呼んでいる．なお，タスク管理の詳細については，後述する．

(3) ジョブ・スケジューラーの機能

第10図に示したようにジョブ・スケジューラーは，ジョブ制御カードを入力データとして読み取り，処理を行う．その処理は，各ジョブ・ステップに対してタスクを生成させるための前準備の仕事である．その主要な仕事はつぎの通りである．

1）入力ジョブの流れを読み取る装置として定められている入力ジョブ装置から，ジョブに関する入力データを読み取る．

2）入力データを解読して，ジョブ制御ステートメントと他の普通の入力データとを見わける．ジョブ制御ステートメントに対しては，その指定の仕方に誤りがないかどうかチェックし，正しければ，その後のオペレーションのために各ジョブの要件を一定の形式にまとめた制御用テーブルを作る．

3）必要な場合には，各ジョブ・ステップに入出力装置を割り当てる．直接アクセス装置の場合には，所要の記憶域の割り当ても行う．両者のすべてが割り当てられたかどうかを確認したうえで，操作員が必要な磁気テープや磁気ディスク・パックを取り付けたかどうかも確かめる．

4）ジョブ・ステップに対しての入出力装置の割り当てが完了すると，タスクを生成し，そのジョブ・ステップで指定されたプログラムの実行を開始する．

5）ジョブ・ステップのプログラムの実行が終了すると，各入出力ファイルについて，指示された後始末を行い，そのステップでの作業の記録（所要時間とか，仕事名など）を出力する．

ジョブ・スケジューラーの一般的機能は以上の通りであるが，ジョブ・スケジューラーにはいくつかのレベルが存在し，それぞれの機能には多少差異がある．ジョブ・スケジューラーのレベルの詳細については，「ジョブ管理プログラム」の章でふれる．

ここでは，一般的にジョブ・スケジューラーが，どの時点にどう機能するかを例を上げて説明しよう．**第12図**を参照していただきたい．この例では，一つのジョブが3つのステップから構成されている．第一のステップでは原始プログラムを言語翻訳プログラムで目的プログラムに変換して，ライブラリー上に出力する．第二のステップで，出力された目的プログラムを連係編集プログラムで実行可能なロード・プログラムにして出力する．最後のステップでは，完成されたプログラムを使って，目的とするデータ処理を行う．

①ジョブ・スケジューラーが主記憶装置に呼び込まれ，第一のステップの言語翻訳のためのジョブ制御カードを読みとって分析する．一定の制御テーブルにまとめ上げたうえで，このステップに要求されている入出力装置の割当てを行い，指示された言語翻訳プログラムの実行のためのタスクを生成する．

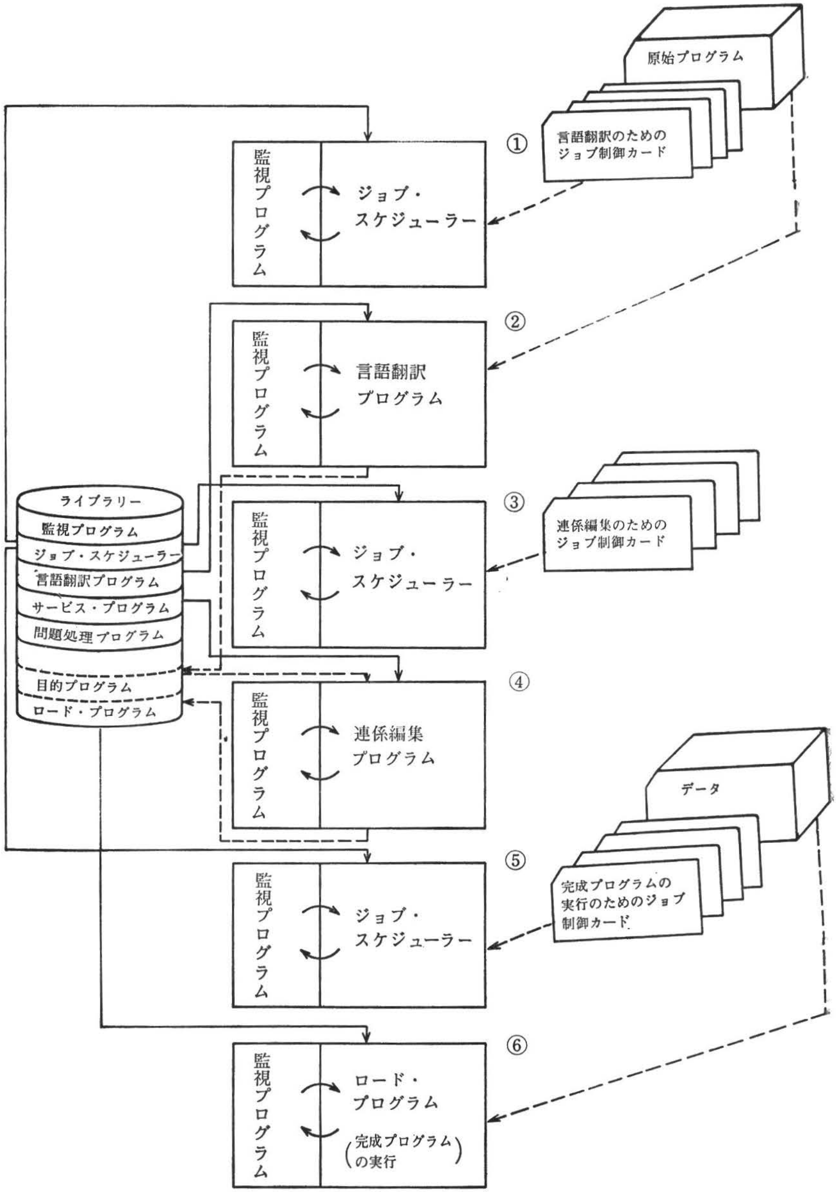
②監視プログラムは，生成されたタスクを制御の単位として，言語翻訳プログラムを実行し，原始プログラムから目的プログラムを作る．

③言語翻訳のステップが終了すると，再びジョブ・スケジューラーがロードされ，前のステップで使用した入出力ファイルの後始末をする．不用になったファイルについては，取りはずしを要求するし，つぎのステップでも使用するファイルはそのままにする．さらに，作業の記録をメッセージとして出力したうえで，つぎのステップのためのジョブ制御カードの解読・分析を行い，入出力装置の準備をする．

④連係編集プログラムが実行され，さきのステップの出力である目的プログラムをロード・プログラムに変える．

⑤③の場合と同様の手順でジョブ・スケジューラーが機能する．

⑥完成されたロード・プログラムが実行され，データの処理が行われる．



第12図　ジョブ・スケジューラーの機能する時点

このようにジョブ・スケジューラーは，ジョブとジョブの間，あるいはジョブ・ステップとジョブ・ステップの間に主記憶装置に呼び込まれ，その機能を果してゆく．このジョブ・スケジューラーの働きによって，いくつかの仕事が中断されることなく，1つの流れとして連続的に処理されてゆくのである．

(4)　マスター・スケジューラーの機能

マスター・スケジューラーは操作員とシステム（監視プログラム，ジョブ・スケジューラーあるいは処理プログラム）間の相互連絡のための橋渡しとしての働きをする．

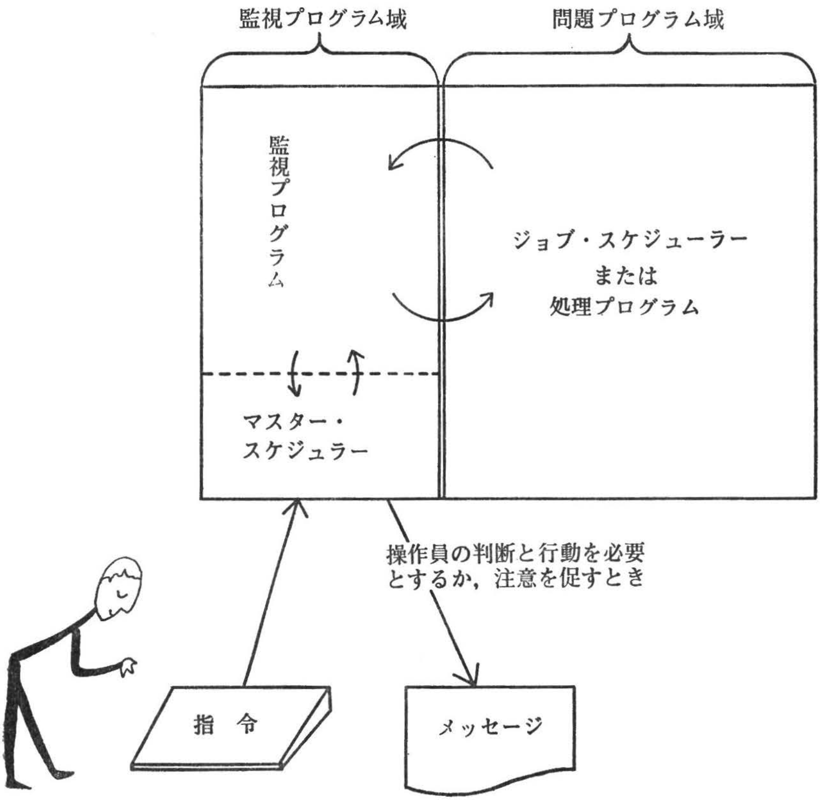
マスター・スケジューラーは，システムが操作員に対して何らかの判断を要求するとか，入出力装置等の操作行動を依頼するとか，あるいは注意を促す必要のあるとき，メッセージを出す．

一方，操作員がシステムに対して，オペレーションの方法を変更したり，システムの状態（現在実行している仕事の名前とか，使用中の入出力ファイルの名前，ディスク記憶装置のスペースの使用状況など）に関する情報を要求したり，あるいは，メッセージに対する応答指令を出すとマスター・スケジューラーを介して処理される．

マスター・スケジューラーの機能を図示したのが**第13図**である．入出力装置としては，タイプライター形式の鍵盤印刷装置や鍵盤装置付映像表示装置などである．後者の場合には，メッセージはブラウン管状の映像装置にうつし出され，操作員の指令は鍵盤装置から入力する．

これまでみてきた制御プログラムの機能は，全体として主要な3つの働きに分かれる．ジョブを連続的に処理してゆくことに関係する**ジョブ管理の機能**(job management)，システム資源を効率的に割り振ってゆくことに関係する**タスク管理の機能**（task management）および入出力データの記録方式と読み書きにかかわる**データ管理の機能**（data management）の3つである．

これ以降，制御プログラムを論ずるさいには，これら3つの範ちゅうに区分して行うことにする．



第13図　マスター・スケジューラーの機能

2.　処理プログラムの構成と機能

オペレーティング・システムのもとでは，何らかの目的としたものを生産するプログラムは，すべて処理プログラムとして定義される．かつ，すべての処理プログラムは，制御プログラムの監督・指導のもとではじめて機能することができる．したがって，メーカーが提供する言語翻訳プログラムやサービス・プログラムはもとより，使用者が作成する問題処理プログラムのすべてが，処理プログラムとして，制御プログラムの配下におかれる．

(1)　言語翻訳プログラム

言語翻訳プログラム (language translator) は，プログラマーが作成した原始プログラムを機械語のプログラム（一般に**目的プログラム**と呼んでいる）に変換するプログラムであることは，今更いうまでもないことである．プログラマーが覚えやすく，書きやすい言語の形式でプログラムを作成することができるのは，この言語翻訳プログラムという仲介役があるからであり，この言語翻訳プログラムによって，プログラマーは，問題解決にさいして，細かでわずらわしい仕事から解放され，効果的なプログラムを作成することができるのである．

現在，使用されている言語翻訳プログラムの代表的なものは，つぎの6種類である：

1）数学的問題の解決を主目的とした言語プログラム：FORTRANとALGOL
2）事務計算的な問題の解決を主目的とした言語プログラム：COBOL
3）FORTRAN と COBOL の両方の特徴をもち，高度な経営計算的問題の解決をねらった言語プログラム：PL/I
4）報告書やファイルの作成に主体をおいた言語プログラム：RPG
5）融通性のある機械中心の言語プログラム：アセンブラー

(2)　サービス・プログラム

サービス・プログラム (service program) は，主としてプログラマーの仕事の負担を軽減するためにサービスするプログラムである．どこの機械室でも必要となる使用頻度の高い，比較的定型化した仕事に関するプログラミングの労をなくするためのサービス，非常に複雑なプログラムを分割して，それぞれに適した言語で作成したサブプログラムを一つのプログラムに連結するサービスなどを行うプログラムである．サービス・プログラムは，連係編集プログラム，分類／組合せプログラムおよびユーティリティー・プログラム群などからなっている．

連係編集プログラム (linkage editor) は，別々に翻訳された複数個の目的

プログラムを，単一の実行可能なプログラムに仕立て上げてくれる．連係編集プログラムを使えば，あるプログラムの全体を翻訳し直さなくても，そのプログラムの中で変更または修正の必要な部分だけを翻訳し直すだけで充分になる．訂正を加えて出来上った目的プログラム部分と残りの目的プログラム部分を結合して完全なプログラムにしてくれる．また，主記憶装置の使用可能な記憶域に収容しきれないような大きなプログラムの場合，そのプログラムを幾つかの部分に分割して，必要のつどオバーレイの形式で主記憶装置に入れるということを可能にするのも連係編集プログラムである．

分類／組合せプログラム (sort/merge program) は，固定長または可変長のレコードを昇順あるいは逆順に並べるための一般化されたプログラムである．分類／組合せプログラムは，独立して使用するのが普通であるが，他のプログラム（たとえば，COBOL 言語で作成したプログラム）の実行中に直接呼び出して利用することもできる．

ユーティリティー・プログラム (utility program) は，次のような機能を果すプログラム群の総称である：

- システム・ライブラリーにプログラムを登録したり，更新したりといったライブラリーの保守機能を果す．
- システム・ライブラリーの中にカタログされているプログラムやデータの索引を印刷する．
- システム・ライブラリーの索引構成を変更する．
- ある入出力装置のデータ・ファイルを別の入出力装置へ移動させる．

このようにライブラリー全般の管理やファイルの転送と復写の機能を行うプログラムの集りが，ユーティリティーというサービス・プログラムである．

第3章

オペレーティング・システムのための ハードウエア構造

一．オペレーティング・システムを前提としたハードウェア

初期のコンピューターは，まず“ハードウェアありき”であった．ソフトウェアにあまりこだわらないハードウェアが設計され，製造されたあとでそのハードウェアの構造に見合ったソフトウェアの開発がなされた．その結果ソフトウェアの作成は非常に困難を極めたし，一定のハードウェアの制約下で行わざるを得ないので利用上の諸要素を十分満足させることのできるものとはなり得なかった．それにハードウェアがもつ諸機能の性能をフルに生かすことにも限界があるのは当然であった．

ところがコンピューターの利用分野が拡大し，その利用法が複雑化してくると，その利用技術であるソフトウェアの開発が出来るだけ容易に行え，かつハードウェアの性能を百パーセント生かすことのできるようなソフトウェアの概念が出てきた．第2世代のコンピューターでは，ハードウェア上の多くの制約を受けながらオペレーティング・システムの構想が序々にソフトウェアに組み込まれていった．その第2世代でのオペレーティング・システムの経験は，新しいオペレーティング・システムへの可能性を追求するための強力なエネルギーとなり，将来のオペレーティング・システムの設計概念へと結実した．

一方で，次の世代のコンピューター・ハードウェアの研究が進んでいたが，それに併行してオペレーティング・システムの哲学ともいうべき設計思想がかたまっていたので，それが新しいハードウェアの方向への有力な手がかりとし

て入力されていったのである．つまり，理想的オペレーティング・システムの設計思想をささえるようなハードウェアの研究開発がなされた．その結果，割込みの機能，記憶保護機能，プログラムの再配置機能，さらに多重プログラミングなどを達成することのできるハードウェアの誕生となった．

したがって，オペレーティング・システムの構造を把握するための前提条件として，オペレーティング・システムをささえるハードウェアの特別な機能を理解しておくことがまず必要となる．ハードウェアにしっかりとささえられたオペレーティング・システムこそが完全なオペレーティング・システムとして機能することが可能なのであり，かつその機能を正確に知るにはハードウェアを無視することはできないからである．そこで本章では，オペレーティング・システムのためのハードウェア構造を検討しよう．

二．割込みの概念

オペレーティング・システムは，コンピューターを休止させることなく仕事を連続的に処理することによってその生産性を最大限にしてゆくことを主目標としている．それゆえに，コンピューター・システムの設計者の最大の関心事の一つは，コンピューターに不測の事態が生じたときでもコンピューターを止めないでそれに対処出来ないかということであった．

一体，コンピューターが事前にキャッチできない不測の事態とはどのような場合であろうか．例えば，つぎのような場合が考えられる．

(1) 入力装置からデータを読み取ったり，出力装置にデータを書き出している最中のエラー，つまり入出力エラーの発生．

(2) プログラマーが適切に指定しなかった命令が実行されたことによるエラー，つまりプログラム・エラーの発生．例えば，6バイトの2つの欄を加えた結果が7バイトを必要とするような命令が実行されたとき．

(3) 操作員がコンソール上の割込みキーを押したとき．この割込みキーは，いづれ押すであろうことを前提に設けられているのであるが，何時，操作員が

押す必要が生じるかは予測できない.

(4)　ＣＰＵ（中央演算処理装置）のある種の構成部品に故障が生じたとき．この種のトラブルの発生は極くまれではあるが，その可能性に対しての考慮は必要である.

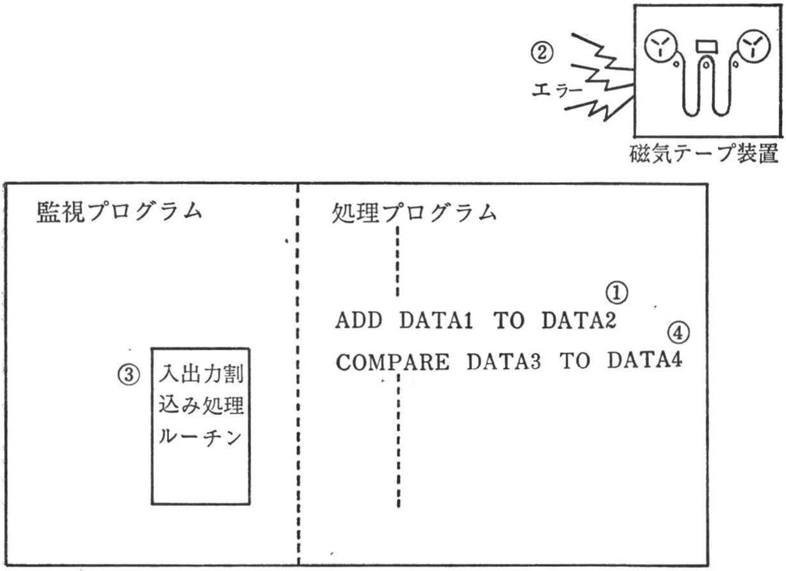
(5)　入出力操作の完了．入出力操作がいったん開始されれば，その完了が，まもなく起るであろうことは予測できるが，正確な完了時点は不測である．入出力によって生ずる待ち時間といえども，ＣＰＵを遊ばせたくないものである.

この種の事態の発生時点の予測ができないことは，コンピューター設計者にとっては大きな関心事である．丁度，電話の利用者にとって，何時，誰から電話がかかってくるかが通常予測できないのと同じ事態である．そこで電話では，呼び手が電話をかけると受信側でベルがなる仕掛けになっている．ベルがなるまでは，目下の仕事を続行できる．ベルがなったときのみ，仕事を一時中断して応答すればよい．続行中の仕事に別の仕事（電話の応答）が割込む状態となる.

コンピューター設計者も，上記のような事態が生じたとき**割込み**（interruption）をひき起すことを考えだした．したがって，オペレーティング・システムの監視プログラムは，不測の事態が何時生じるかをたえずチェックしつづける必要はなくなる．ＣＰＵを占有して仕事を行っているプログラムにとってみれば，不測事態が生じたときのみ割込みの状態が発生することになって，一時そのプログラムの実行が中断されることになる.

では，一体オペレーティング・システムの制御プログラムはどのようにして“割込み”に対処するのであろうか．先述のいくつかの例をとり上げて説明しよう.

第１図は，入出力エラーによる割込みが生じた場合の例である．ＣＰＵが入出力エラーの割込みを認知すると，ＣＰＵの制御権は，監視プログラムの一部を構成している**入出力割込み処理ルーチン**に移る．つまり，図において①の命令の実行中に，磁気テープからの読込みエラーが生じたとする（②）．このエラーをＣＰＵが感知するとすぐに，③のルーチンの最初の命令が実行される.

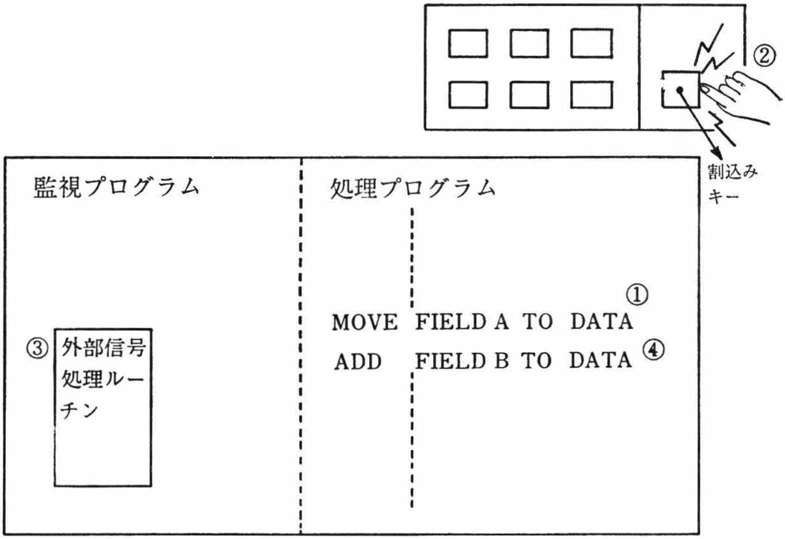


第1図 入出力エラーによる割込みとプログラム実行の流れ

このルーチンでは，エラー・レコードに対応するテープ上のレコードを所定回数だけくり返し読み直す．どうしても正しく読み込めない場合には，プログラム上の指示にもとづいた処理をしたうえで，再びCPUの制御権は問題処理プログラムにもどされる（④）．見かけ上は何ら中断は生じなかったと同じように問題処理プログラムの実行がつづく．

入出力の操作が完了した場合の割込みも上述の制御権の移動と同じである．入出力エラーや操作の完了という事態に生じる割込みを総称して**入出力割込み**（input/output interruption）と呼んでいる．

第2図に示したのは，操作員が何らかの手操作の必要のためコンソール上の割込みキーを押した場合の流れである．①の命令を実行している最中にキーが押されたと仮定する（②）．この割込みをCPUが感知すると即座に制御権は，監視プログラムの中の**外部信号処理ルーチン**に移り，そのルーチンの最初の命令から実行される．このルーチンでは，コンピューターを一時「**待ち状態**」に

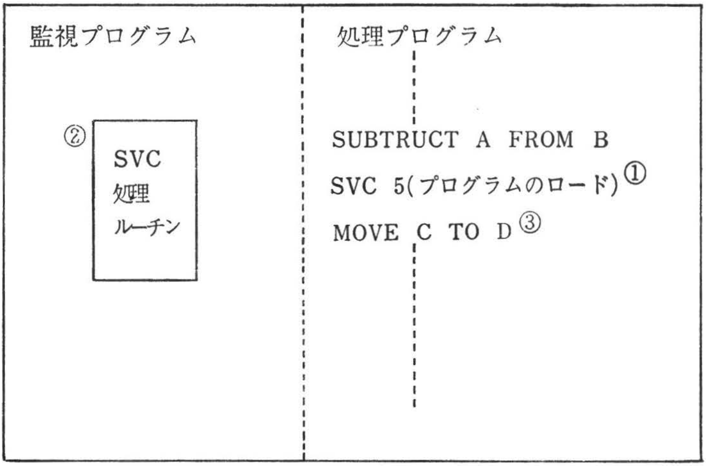


第2図　割込みキーによる割込みとプログラム実行の流れ

おいて，操作員の操作を可能にする．操作員が必要な処理を終えて再び割込みキーを押すと「**作動状態**」にもどり，④へと制御権が移る．このような外部からの信号にもとづいて発生する割込みを**外部割込み**（external interruption）と呼ぶ．

プログラマーが，プログラム言語の約束に違反した命令の指定を行ったり，不適切な命令指定を行った場合にも，前述と同じような手順で制御権が監視プログラムに移ってゆく．言わば誤りのあるプログラムなので多くの場合は，そのプログラムの実行は放棄して（勿論，誤りの原因をつきとめるのに必要となる情報は印刷したうえで），つぎの仕事の処理のためのプログラムに移る．この種の割込みは**プログラム割込み**（program interruption）という．

機械の機能上のエラーによって生じる割込みは，**機械チェック割込み**（machine check interruption）と呼ばれる．機械チェック割込みが生じると，やはり監視プログラムの中の対応する処理ルーチンがＣＰＵの制御権を受けとり，その原因を診断する．診断結果が致命的なものであれば，機械は「**待ち状態**」に入ってしまう．もし，作動続行可能であると判断した場合には，診断結果を



第3図 監視プログラム呼出しとプログラム実行の流れ

後の保守作業に活用出来る形式で外部記憶装置（磁気ディスクや磁気ドラム）に記録したうえで，処理プログラムに制御権をもどす．

以上の4種類の割込みは，いずれも不測の事象が発生したことにもとづくものであるが，プログラムの命令で割込みを要請することもできる．この割込みは**監視プログラム呼び出し割込み**(supervisor call interruption)と呼ばれる．その名前が示すように主として問題処理プログラムで，監視プログラムの機能を利用したいときに使う命令である．たとえば，あるプログラムの実行が完了したので，その仕事の後始末をしたうえでつぎのプログラムの実行に移って欲しいとか，サブプログラムのローディングを要求するとか，入出力操作の要請をするとかいった場合には，「**監視プログラム呼び出し命令**」を使って割込みを起させる．この命令コードは「**SVC**」であり，そのオペランドに要求する機能を表示するコードを記入する．

第3図にその例を示した．①の SVC 命令が実行されると直ちに割込みが生じる．②の SVC 処理ルーチンが CPU で実行され，SVC のオペランドで指示された要求機能を果すルーチンに移って所要の処理をしたうえで③にもどる．

今までの説明では，割込みは一見通常のブランチと同じように理解される可能性があるが，割込みはブランチとは根本的に異なる．割込みは単にプログラムの実行順序が一時変更されるだけでなく，CPU の状態自体がハードウェア的に切り換えられるのである．さらに，監視プログラム呼び出し割込み以外は，不測の事態を CPU が感知することによって自動的に生じるものである．以下割込みに関係するハードウェアの構造についてふれてゆくことにする．

三．プログラム状況ワードと割込み

プログラム状況ワードは，プログラムを正しく実行してゆくために必要となる情報が集約されている８バイト（倍ワード）のレジスターまたは記憶域である．そこには，次に実行すべき命令が記憶されている主記憶装置のアドレス，条件コード，命令の長さなどが示されている．

プログラム状況ワードは，一般に PSW（Program Status Words の略）と呼ばれており，その目的は，命令の実行順序を制御し，実行中のプログラムに関係するハードウェア・システムの状態を示したり，その状態を保持することにある．この PSW の形式を示したのが**第4図**である．下側に示した数字は，PSW を構成する各種情報のビット位置である．それぞれの情報の意味は後述する．

システム・マスク	保護キー	AMWP	割込みコード	命令長コード	条件コード	プログラム・マスク	命令アドレス
ビット 0–7	8–11	12–15	16–31	32–33	34–35	36–39	40–63

第4図　プログラム状況ワード（PSW）の形式

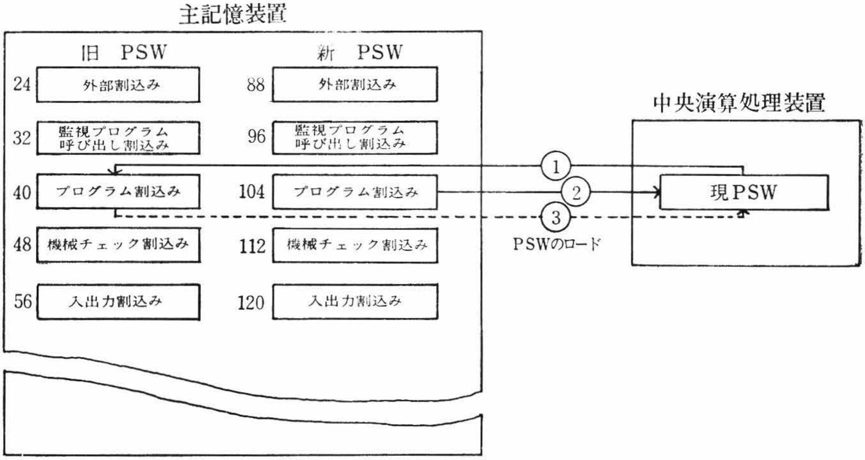
すでに説明したように，CPU が割込みを感知すると，直ちに，その割込みに対応する特定の割込み処理ルーチンの最初の命令が実行される．割込み処理ルーチンの最初の命令が実行されるための前提条件は，CPU がその最初の命令アドレスを知ることである．五種類の割込みが存在することからすれば，それぞれの割込み処理ルーチンの最初の命令のアドレスを提供するためには，目

下プログラムの実行を制御している PSW に置き替るべき5つの PSW が用意されておることが必要となろう．割込みによってとって代る PSW を一般に「**新 PSW**」と呼んでおり，これらは主記憶装置の特定の場所におさめられている．いま入出力割込みが生じたとすると，CPU にある PSW（これを「**現 PSW**」と呼んでいるのだが）に置き代って入出力の新 PSW が現 PSW になる．つまり，CPU の PSW レジスターに自動的にロードされる．この新 PSW におさめられている命令アドレスこそが，入出力割込み処理ルーチンの最初のアドレスなのだ．

CPU が，発生した割込みのタイプ（プログラム割込み，入出力割込み，機械チェック割込み，外部割込み，または監視プログラム呼び出し割込みのいずれか）を識別するやいなや，CPU に登録されている現 PSW は，自動的に貯えられる．現 PSW が一時保持されていないと割込み処理ルーチンでの処理が終了したあと再び元のプログラムの中断点の次の命令にもどる手がかりを失ってしまうことになるからだ．CPU にある現 PSW が追い出されて主記憶装置に貯えられたものを旧 PSW と呼ぶ．現 PSW が旧 PSW に追い払われたうえで新 PSW が CPU の PSW レジスターにロードされて，この新しい PSW の命令アドレスにしたがってプログラムの実行が再開される．実は，この現 PSW の追い出し，新 PSW のロードというハードウェア機能が「割込み現象」なのである．この割込みという現象は全く自動的に達成されるものであり，特別な時間がかかるものではない．

現 PSW，旧 PSW および新 PSW の相互関係と割込みによるプログラムの切り換わりの状況をまとめたのが第5図である．新 PSW と旧 PSW は，各々割込みのタイプに対応する5種類のものが主記憶装置の所定のアドレスに確保されている．各 PSW の左側に示した数字がそのアドレスである．目下実行中のプログラムの実行順序やそれに関係するシステムの状況を保持しているのが現 PSW であり，これは CPU 内の特別なレジスターに入っている．

第5図ではプログラム割込みが生じたさいにどのように PSW が置きかわるかを示している．まず，現 PSW がプログラム割込みの旧 PSW に貯えられる（①）．そのあとで，プログラム割込みの新 PSW が現 PSW にとって代る

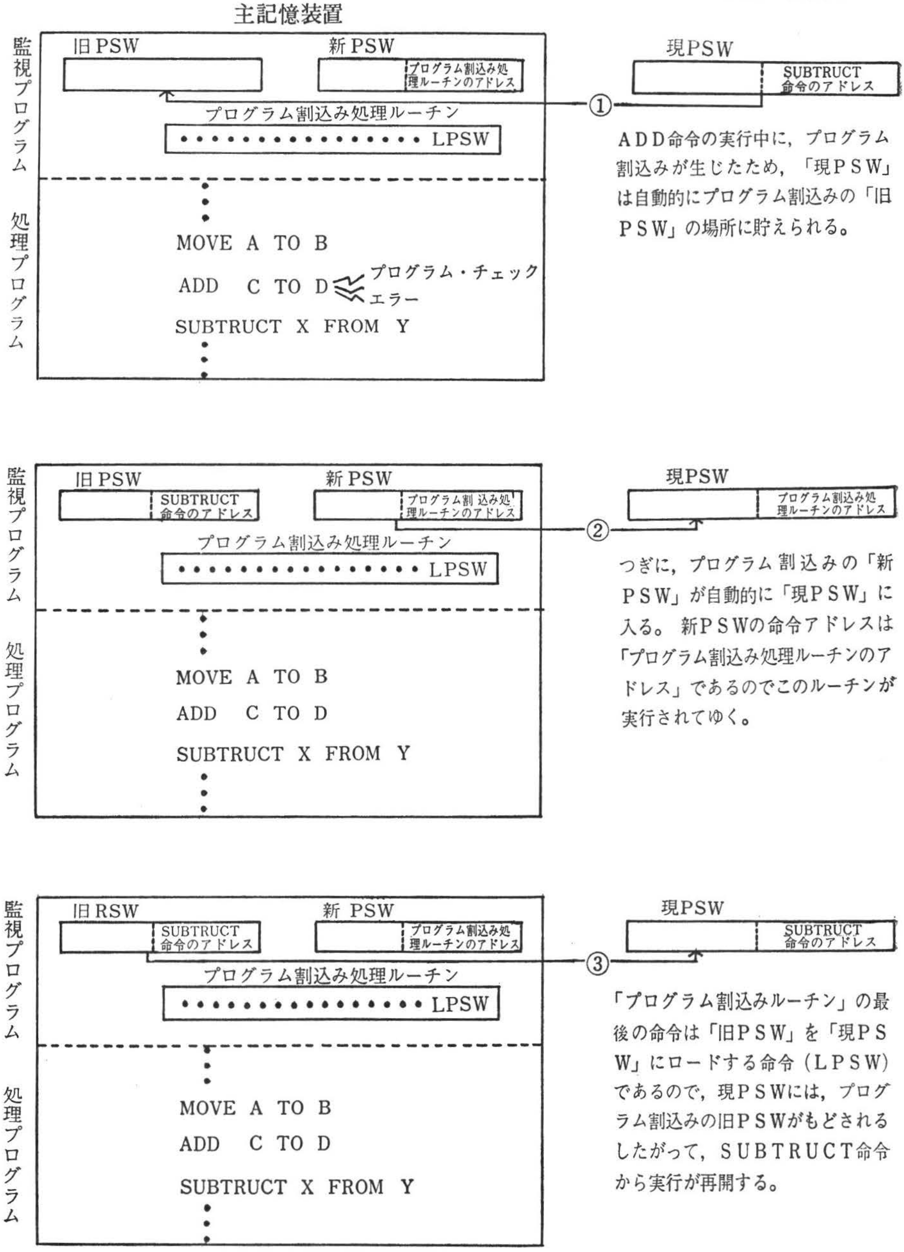


第5図　割込みによるプログラムの切り換え

(②). このように新，旧の PSW が交替する現象が割込みというハードウェア機能である. 割込みが生じると新 PSW の命令アドレスで示されている命令から実行が開始される. この命令のアドレスが割込み処理ルーチンのアドレスである. 割込み処理が終了すると，元のプログラムの中断点から処理を再開する必要が生じる. そのためには，割込みが生じたとき貯えられた PSW，つまり，旧 PSW を現 PSW にもどせばよい. これを行うために用意されている命令が「LPSW」である. 第5図で，破線で示した流れがそれである. 第5図で示したプログラム割込みの例を，監視プログラムと処理プログラムの関係で具体的にまとめたのが**第6図**である.

他の割込み，つまり，入出力割込み，外部割込み，機械チェック割込みおよび監視プログラム呼び出し割込みのいずれの場合にも同様の実行順序で処理されてゆく.

四．システム状態の切換え



第6図　プログラム割込みによるプログラム実行順序

四．システム状態の切換え

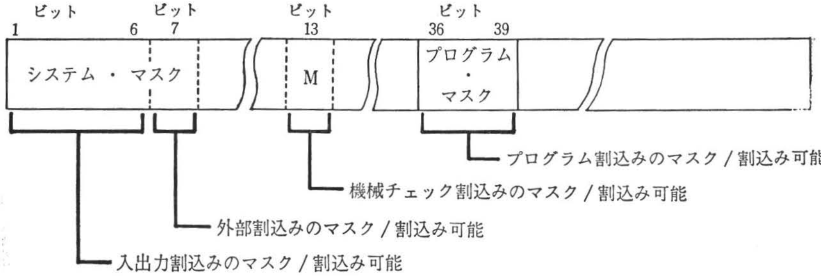
1.　マスク状態と割込み可能状態

いま，同時に2個の入出力割込みが生じた場合を仮定してみよう．ある入出力割込みが生じた直後に別の入出力割込みが生じるといった事態である．2番目に生じた割込みの方がCPUを占有してしまうことになる．このような場合が許されるとしたら，割込み処理ルーチンで最初の割込みにどのような措置を加えるべきかわからなくなってしまう．なぜなら，最初に生じた割込みによって貯えられた旧PSWが二番目の割込みで破壊されてしまうからだ．

もし，最初の割込みを処理しているときに生じた2番目の割込みが全く無視されてしまうようなハードウェアにしておいたらどうなるだろう．ある割込みの直後に生じた割込みは永久に処理出来ないという不合理が生じる．そこで，第3世代のコンピューターでは，入出力割込みや外部割込みに対して特定の割込みを処理している間に生じた同種の割込みを一時的に無視し，留保しておくことができるようにした．一方，機械チェック割込みとプログラム割込みのうちの加減乗算で生じる桁あふれに関しては必要に応じて割込みを全く禁止してしまうことができるようにした．これらのことを**マスク状態**（masking state）という．

このように一時的にあるいは永久に割込みを無視してしまうマスク状態は，PSW の特定のビットをゼロにセットすることによって作り出される．逆に，現PSW のマスク・ビットが1にセットされているときには割込みが可能になる．

マスク状態と**割込み可能状態**（interruptable state）との切換えに使われるマスク・ビットの位置をまとめたのが**第7図**である．監視プログラム割込みとプログラム割込みのうちマスクできないものは，常に割込み可能状態にある．マスクの切換えの方法としては，PSW 全体を入れ替えることによって(LPSW 命令で PSW をロードするか，SVC 命令で強制的に割込みを生じさせて PSW



第7図　マスクと割込み可能情態

を取り替える),新しいマスク・ビット構成の PSW にするか，特別な命令でマスク・ビットの一部を切り換える方法がある．プログラム・マスクは，**「プログラム・マスク設定命令」**によって現 PSW のその部分だけを変えることができる．一方，システム・マスクは，**「システム・マスク設定命令」**によって変更できる．機械チェック・マスク・ビットだけは新しい PSW 全体を入れ替えることによってのみ変えられる．

この種の切替えは，制御プログラムが柔軟性のあるシステム管理を行うために工夫されたものであって，われわれが問題処理プログラムを作成するにさいして全く考慮を払う必要はない．しかし，オペレーティング・システムの構想からすれば，欠くことのできないハードウェア機能なのである．

2. 作動状態と待ち状態

ＣＰＵの処理速度と入出力処理速度を比較した場合，とかく入出力操作が遅れがちである．入出力操作にＣＰＵの処理を同期化させるようなハードウェア・システムにしてしまったとしたら，入出力処理速度の限界によってＣＰＵの処理速度が制約されてしまうであろう．

ハードウェア設計のかなり初期の段階から，入出力操作と CPU 処理が同時併行的に行いうるような工夫はなされてきたが，入出力操作に CPU 処理が依存する形式での併行処理には限界があった．ところで，ハードウェアを効率よくこき使う立場にあるオペレーティング・システムの観点からすれば，いっそのこと CPU と入出力操作との同期化の仕事を制御プログラムに委譲してくれ

た方がよいということになる．つまり， CPU から入出力操作の要求を出したあと，その操作が完了するまでの待ち時間を制御プログラムで自由に管理することができるとしたら，その時間を有効に他の仕事にふり向けることが出来るからだ．ただし，何らかの方法で入出力操作が完了したことがシステムに知らされることが必要となる．そこで，先述のように「入出力操作が完了した」という事態を割込みによって感知する方法がとられた．

このような考えに立脚すると CPU に生じるであろう待ち状態を完全にソフトウェアの管理下におくことになる．

そこで CPU に何らかの処理を行わせることと，一時他の事象が完了するまで待たせておくということをハードウェア機能を使ってソフトウェアが管理できるようにする必要が生じる．この要請を満すために PSW のビット位置14が設けられた．

このビットが1の場合， CPU は**待ちの状態**（waiting state）におかれる．待ち状態では命令は処理されないし，したがって，主記憶装置からデータがとり出されることもない．ただし，待ちの状態は停止状態とは異なる．停止状態では入出力操作も CPU 処理も停止しているが，待ち状態では CPU の処理が中断しているだけで，入出力操作は行われているし，割込みも可能である．

一方このビットが0の場合が，CPU の**作動状態**（operating state)である．この状態では，命令の取り出しと実行が正常な形で行われる．

待ち状態と作動状態の切換えは，完全に PSW を入れ換えることによってのみ行われる．したがって，作動状態から待ち状態に切換えるには， LPSW 命令あるいは監視プログラム呼び出し命令を出すか他の割込みによるしかない．一方，一たん待ち状態に入ったものは入出力の完了か外部割込みによって切換えが達成される．

主記憶装置にある新旧 PSW は，監視プログラムの意図通りに変更できるし，LPSW 命令や SVC 命令も使えるということになると，CPU の待ち状態と作動状態の管理を完全に制御プログラム下におくことができるのである．

3.　監視プログラム状態と問題プログラム状態

オペレーティング・システムをささえるためのハードウェアは，その便宜を計るために種々の特殊な機能を制御プログラムのために解放する．それらの機能の多くは，制御プログラムのためにだけ必要であって，一般の処理プログラムの作成過程では何ら関係しない．さきにみた LPSW 命令は，制御プログラムがプログラムの実行順序を定め， CPU の状態を切り換えるために必要となる命令であって，一般のプログラムで必要なものではない．もしも，処理プログラムで自由に PSW を変えられてしまったのでは，制御プログラムでシステムを統一的に管理出来なくなるばかりでなく，間違った使い方をされてしまってシステムを混乱に落し入れてしまう可能性がある．たとえば，誤って CPU を待ち状態にしてしまって作動状態にもどせなくしてしまうことだって考えられる．

したがって，ある種の命令は一般のプログラマーが使うことを禁止するようなハードウェア措置が必要となってくる．この一般に使用させないための命令を**特権命令**（privileged instruction）と呼んでおり，監視プログラム状態でのみ実行できる命令である．PSW のビット位置15が 0 である場合 CPU は**監視プログラム状態**（supervisor state）にあり，特権命令も実行することができる．一方，このビットが 1 であると**問題プログラム状態**（problem program state）であり，特権命令を実行しようとすると特権命令違反となりプログラム割込みが生じる結果となる．

第 1 表に示したのが代表的な特権命令である．この表から判るように，問題プログラム状態では，入出力命令，記憶保護命令や診断命令が PSW ロードやシステム・マスク設定命令と同じように無効となってしまうのである．監視プログラム状態は明らかに制御プログラムのために設けられた状態であるから，通常のプログラムでは入出力命令は使うことはできないという奇妙な事態になる．ところが，問題処理プログラムの中では，必ず GET，PUT とか READ，WRITE のたぐいの入出力命令を使っている筈である．実は，この種の命令は機械語の命令に翻訳されると入出力命令になるのではなく， SVC 命令に代るのである．この SVC 命令によって監視プログラム呼び出しの割込みを生じさせる．その結果，監視プログラムの中にある入出力処理ルーチンを使う．入出

第1表 主要な特権命令

特権命令	命令コード	機能
PSWロード	LPSW	現PSWを記憶装置からロードする
システム・マスク設定	SSM	現PSWのシステム・マスク・ビットの切替え
記憶キー設定	SSK	所定の記憶域の保護キーをセットする
診断	(83)	CPUの正しい働きの確認とエラーのある部分のつきとめ
入出力開始	SIO	指定された入出力装置の操作開始の要請
入出力テスト	TIO	指定されたチャネル，入出力装置のテスト

力命令を特権命令としているのは，入出力関係のルーチンが非常に複雑で書きにくいという理由もさることながら，入出力処理を制御プログラムで統一的に支配することによって，入出力によって生じる待ち時間を有効利用することでシステム全体の効率を高めようという意図である．

PSW のビット位置15を変えることによって，問題プログラム状態と監視プログラム状態間での切替えを行うことができるのであるが，このビットは，新しい PSW を現 PSW に置き代えることによってのみ変えることができる．したがって，状態の切換えは，PSW ロード命令で，ビット位置15に所要の値をもった新しい PSW を入れることによって行う．ところが，PSW ロード命令は，特権命令であるから監視プログラム状態においてのみ切換えが可能となる．もちろん通常の割込みによって PSW が入れ替わることで状態が切換わることは考えられる．

監視プログラムから処理プログラムに制御権がもどされるさいには何時でも問題プログラム状態にされている．一方，割込みによって制御権が監視プログラムに移ったさいには必ず監視プログラム状態になっている．このような方法によって，制御プログラムが実行されるさいの状態と問題処理プログラムが実行されるさいの状態を区別することで，制御プログラムにのみ解放した特権が一般のプログラムの中で乱用されて害をおよぼすことを防止しているのである．

五．記憶保護機能

オペレーティング・システムでは，必ず監視プログラムと処理プログラムが主記憶装置に同居する．監視プログラムの中核をなす部分は常駐監視プログラムとして何時でも主記憶装置にある．他の監視プログラム部分は必要のつど主記憶装置に呼びこまれる非常駐監視プログラムである．一方処理プログラムは，仕事が処理されるごとに次から次へと移り変ってゆく．監視プログラム自体は，充分にテストされて完成されたプログラムであるから，まず誤りを犯すことはないとみてよかろう．ところが処理プログラムでは，不完全なプログラムもあるし，目下テスト中の誤りだらけのプログラムだってある．となると，処理プログラムが誤って監視プログラムの記憶域にデータを移動したり，そこで演算を行って監視プログラムの一部を破壊するおそれは十分にある．あるいは逆に監視プログラム域の情報を取り出して処理に使う可能性もある．

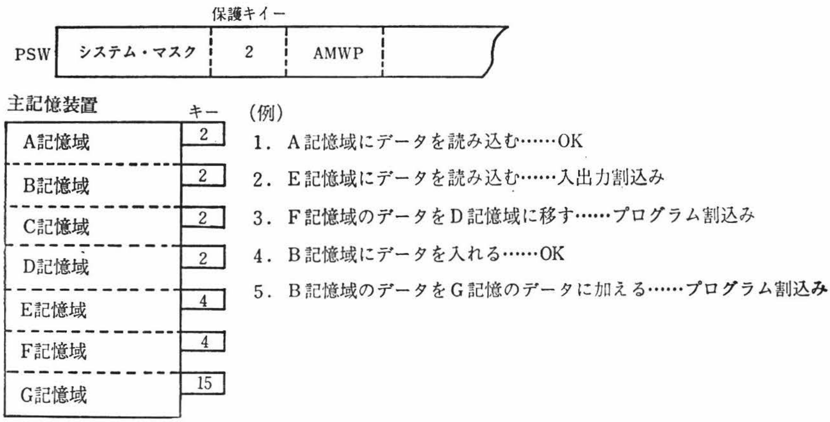
監視プログラムと1つの処理プログラムが同居した場合よりも，監視プログラムと複数個の処理プログラムが同居するという多重プログラムの状況下ではますます深刻な問題となってくる．

当然のことながら，オペレーティング・システムをささえるハードウェア機能として何らかの方法で他のプログラム域を侵害しないようにする必要がでてくる．われわれが日常生活で他から侵入されないようにするための防禦措置として鍵をかけると同じような構想を主記憶装置にも適用せざるを得なくなる．記憶域に鍵をかけてしまえば，合鍵をもたないかぎり，書き込みも取り出しも出来ないはずである．ところで，主記憶装置に1つの鍵をかけてしまったのでは意味がない．監視プログラム域と処理プログラム域に別々の鍵を，さらに多重プログラミング下では，それぞれの処理プログラム域に異った鍵を必要とするだろう．

そこで，主記憶装置をいくつかのブロックに分けブロック単位に鍵をかけることが出来るようにする必要が出てくる．その結果，主記憶装置を2K（1Kは1024バイト）単位に1つのブロックとし，各ブロックに鍵にあたる番号を記

憶できるようにした．番号を記憶する記憶域は通常の主記憶装置とは別に設けられており，5ビットからなる．5ビットのうちの最後のビットは，書き込みと取り出しの両方を保護するか，書き込みのみを保護するかを示すのに使い，上位4ビットで鍵番号を示す．4ビットであるから0〜15までの16種類の鍵が区別出来る．

一方，合鍵に当るものはどこに設定されるのであろうか．実は，この合鍵に相当するものがPSWにある．**保護キー**（protection key）という言葉で示されている4ビットの部分である（第4図参照のこと）．この保護キーに保持されている数字と記憶ブロックの鍵番号（通常，単にキーと呼んでいるが）が合致したときのみ，そのブロックに書き込みあるいはブロックからの取り出しが許される．目下のPSWの保護キーと一致しないブロックに書き込むとか，そのブロックから取り出すといった違反を犯すとプログラム割込みが発生してしまう．



第8図　記憶保護機能の構想

第8図にいくつかの例をあげて，保護機能を図示した．各ブロックにキーをセットする命令が，「**記憶キー設定**」命令である．あるブロックにどのようなキーがセットしてあるかを調べる命令が「**記憶キー挿入**」命令であり，いずれ

も特権命令である．したがって，これらの命令は監視プログラム状態でしか実行できない．ということは，この保護機能は監視プログラムによって全て管理されるということを意味している．なお，保護キーが0であることは，マスター・キーを意味している．つまり，現 PSW の保護キーが0であれば，各記憶ブロックのキー番号がいくつであっても無関係にどのブロックに対しても書込みと取り出しが許される．この0という保護キーは監視プログラムが実行されるときに必要である．なぜなら，監視プログラムは，処理プログラムからある種の処理要請をうけたとき，その処理のもとになる情報を処理プログラムが記憶されているブロックから取り出したり，逆に処理結果をそのブロックにもどしたりするからである．

このようにオペレーティング・システムはハードウェアの保護機能を採用して，システム全体の信頼性を向上してゆくことに努めているのである．

六．割込み処理ルーチンの概念

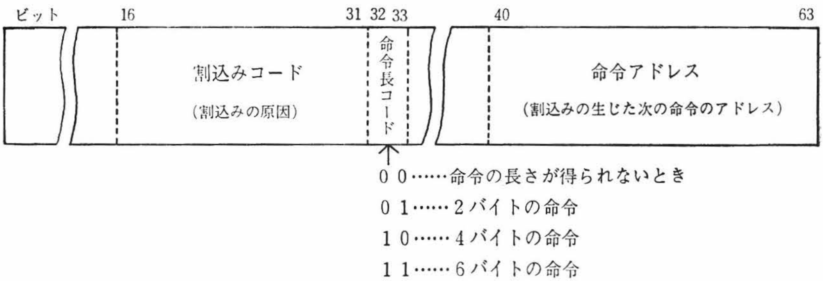
割込みが生じた場合には，その割込みを適切に処理するルーチンが必要である．起りうる全ての割込みを処理するためのそれぞれのルーチンが監視プログラム内に用意されている必要がある．5種類の割込みに対しておのおのの新 PSW があるので，この新 PSW の命令アドレスに対応する割込み処理ルーチンの最初のアドレスを挿入しておけば，ある種の割込みが生じると無条件にその割込みに見合った処理ルーチンに制御権が移る．

それぞれの割込みルーチンでは，何の原因で割込みが生じたのか分析する必要がある．割込みが生じた理由は旧 PSW のビット位置16から31に貯えられている．この16ビットからなる場所は「**割込みコード**」と呼ばれている．入出力割込みの場合には，どのチャネルのどの装置が原因となって割込みが生じたかが示される．プログラム割込みの場合には，割込みの詳細な理由が示される．たとえば，問題プログラム状態で特権命令を使ったとか，記憶保護違反をおかしたとか，四則演算の結果桁あふれが生じたとかの理由がコード化して示される．

監視プログラム呼び出し割込みでは，ＳＶＣ命令のオペランドで示したコードがそのまま割込みコードに入る．外部割込みでは，割込みキーを押したとか時限装置（タイマー）に指定した時間が経過したとかの理由が示される．

それぞれの割込みの処理ルーチンでは，この割込みコードを分析することで原因をつきとめ，それに見合った処理ルーチンにブランチさせる．

プログラム割込みのような場合には，割込みの原因となった命令の場所をつきとめる必要が出てくる．旧 PSW には，割込みの生じた命令のつぎのアドレスが入っている．したがって，その命令（割込みの原因となった命令）の長さがわかれば，命令の正確なアドレスがつかめる．**第9図**に示しているように，実は割込みが生じたときの命令の長さも旧 PSW に保持されている．



第9図　割込み処理ルーチンで必要となる旧 PSW の情報

旧 PSW の中にある命令アドレスから命令長コードで示されている命令の長さを差引けば正確なアドレスが得られるのである．このコードは，割込みが生じたときの命令の長さが不明なときには00となっている．

このようにして，割込みの原因をつかみ，必要なら割込みを引起した命令のアドレスをとらえたうえで，適切な処理を行う．ある場合には，最早その処理プログラムの続行は無意味とみなして，つぎの処理プログラムに移る場合もあるし，またはそのプログラムの実行を一時中断して，別の処理プログラムに制御権を渡すという場合（多重プログラミングのとき）もある．何ら支障がないと判断した場合には，旧 PSW をロードすることによって元の処理プログラムの中断命令のつぎにかえって処理を再開することになる．

このような構想にもとづいて，あらゆる割込みに関してのそれぞれの処理ルーチンが確立され監視プログラムの中に組み込まれているのである．

七．プログラムの再配置構造

1. プログラム再配置の概念

プログラマーは，プログラム作成過程で記号を使ってデータ記憶域を参照する．言語翻訳プログラムによって機械語のプログラムに変換されると，これらデータ参照記号やブランチ先に使った記号は，すべて主記憶装置の特定域をさす番地におき替えられる．このことは，今更いうまでもないことであろう．この機械語化された番地は，固定化された絶対番地であるのが普通だ．言語翻訳の段階で，そのプログラムがどの位置からロードされるかが決定してしまう．それにともなって，個々のデータの記憶域や命令のブランチ先なども固定化される．このように絶対番地が定められたプログラムでは，そのプログラムの実行段階で主記憶装置のあいた場所を自由に選定してロードすることは不可能である．主記憶装置の選択がきかないプログラムとなってしまう．

少し融通性をもたせるやり方として，連係編集プログラムの段階で絶対番地を決定する方法がでてきた．言語翻訳プログラムで変換された機械語プログラムに含まれる番地は相対番地として設定される．連係編集プログラムの主要目的は，別々に翻訳された機械語プログラムを結合して，１つの完成したプログラムにすることにある．無関係に翻訳された個々のプログラムに絶対番地が割り当てられ動かしがたいものであったら，結合のしようがない．Ａというプログラム部分の一部とＢというプログラム部分の一部が，同じ記憶域にロードされるようになっておれば，両者が重複しないように番地を動かす必要があろう．連係編集プログラムの機能に，サブプログラムやサブルーチンの結合機能がもたされている以上，このプログラムへの入力である機械語プログラムの番地は可変性のある番地，つまり相対番地である．連係編集プログラムは，各サブプログラムやサブルーチンの主記憶装置に占める領域を定めたうえで，それぞ

れのサブプログラムやサブルーチンに含まれている番地部分に一定の数字を加えることによって相互の調整を計ったうえで絶対番地を作りだす．

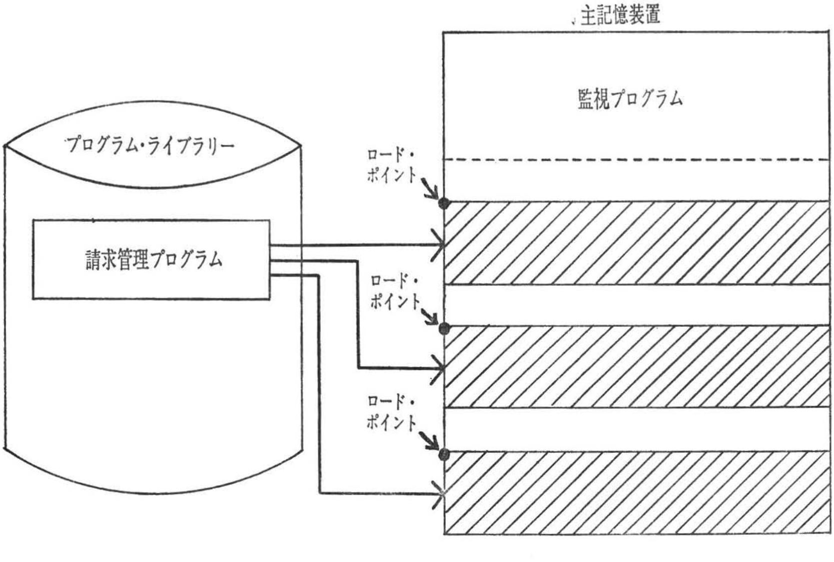
この連係編集プログラムで決定した番地が固定番地となったのでは，プログラムの再配置は充分行なえない．もしも，連係編集プログラムで設定した記憶域以外の場所にそのプログラムをロードしたい要請がでてきたら，再び連係編集プログラムで番地調整の必要がでてくる．つまり，連係編集プログラムの機能を使って再配置を行うことは，あまり効率的とはいえない．もっと動的に再配置が可能になることが要求される．**動的再配置**（dynamic relocation）が要求される主要な理由は，つぎの2点にしぼられよう．

まず，多重プログラミングの状況下で主記憶装置を効率的に使うことにある．多重プログラミングとは，一度に複数個のプログラムを主記憶装置に共存させて，CPU を共用させながら，並行的に処理してゆくことにある．もし，連係編集プログラムによって各々のプログラムの記憶域が固定的に割り当てられるとしたらどうであろう．あらかじめ主記憶装置を固定的に分割しておき，このプログラムは，この区画でのみ実行するといった関係づけを行ったうえで連係編集する必要が生じる．それぞれの区画は，かなり大きなプログラムをも収容し得る記憶域を見込んでおかなくてはなるまい．どんな小さいプログラムが実行されるときでも，その区画は占有されるだろう．また，ジョブが上手にスケジューリングされていないと，空いた区画にロードして実行するプログラムが存在しないということになりかねない．遊んでいる記憶域をみすみす無駄にしてしまうことになる．主記憶装置の使用効率を上げ，かつジョブのスケジューリングの負担を最少限にするためには，プログラムの実行直前に主記憶装置の空いている場所を認知し，その場所にロードして実行できることが必要である．

動的再配置が要求される別の理由は，つぎのことにある．近年，データ通信をはじめとした複雑なプログラムが出てきた．1つのプログラムの中でいろんな要素の処理が要求される．全ての要素を満すプログラム全体を主記憶装置にローディングしておくには，プログラム自体が大きすぎるし，また使用頻度の低いルーチンまで常時記憶させておくのは無駄が多い．さりとて，オーバレイ

の手法を使うこともむずかしい．なぜなら，連係編集プログラムでオーバレイ構造を作らせるための前提は，あらかじめプログラムの実行順序が十分予期できることにある．オーバーレイ・ツリーのようなものが組み立てられるときに効果的に応用できるのである．ところが，プログラムが複雑化し，例外要件などがふえてくるとなかなか実行順序が予知できなくなるし，また，プログラムの各所から例外要件ルーチンを参照することもでてくる．このようなプログラムになってくると，プログラムの実行過程で特殊要件が生じたときだけ，そのルーチンを主記憶装置の未使用の場所に動的に呼び出して実行できることが強く要求されるのである．主記憶装置の空きの場所にダイナミックにロードできることが必要となるので，どうしてもプログラムの動的再配置機能がのぞまれる．

ところで，実行直前に主記憶装置の空いている場所を見つけ出し，プログラムに含まれている番地を調整するにはどうしたらよいだろう．主記憶装置の使用状況を監視し，どの場所が未使用の状態にあるかをつきとめる働きは，監視プログラムにある．監視プログラムは，プログラムのローディングの必要性を



第10図　プログラムの再配置

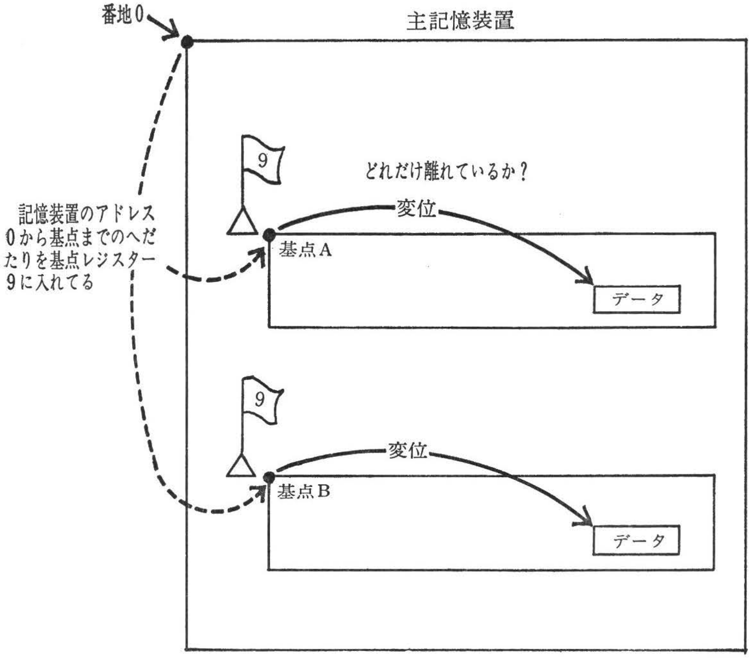
感知すると，主記憶装置の空いている場所をそのプログラムに割り当てる．したがって，あるプログラムが実行される時点が異なると，主記憶装置の異なった場所が割りふられる．たとえば，**第10図**を参照していただきたい．プログラムが使用される時点が異なるとそれぞれ異なった場所に再配置の形式でローディングされる．監視プログラムは，ロード・ポイントを定めて，そこを基準にしてプログラムをロードしてゆく．もし，監視プログラムが，**ロード・ポイント**（load point）をもとにプログラムに含まれている全ての番地の調整を行いながらローディングするとなると極めて効率は低下するし，機械語命令の連続する中から番地部分を見つけ出すこともむずかしい．

当然の方向として，ハードウェアの機能で簡単に相対番地を絶対番地に変換できないかということになる．そこで，基点レジスターと変位による番地決定という設計構想が生まれた．

2.　基点レジスターと変位による番地の決定

プログラムが機械語に翻訳されるとき，データ記憶域やブランチ先を指示する記号番地が，絶対的番地になるのではなく，**基点レジスター**（base register）と**変位**（displacement）という形式をもった相対的番地となる．命令がCPUで解読され実行されるときに基点レジスターの内容と変位そのものが加算されることによって得られた数値が有効番地として使われる仕組みである．したがって，基点レジスターの内容を変えておきさえすれば，異なった位置を指すようになる．**第11図**をみていただきたい．この例では，ある種のプログラム部分に割り当てられた基点レジスターを９と仮定している．同じプログラム部分に対して，ロード・ポイントにAが割り当てられる場合とBが割り当てられる場合を示している．ロード・ポイントにAが割り当てられる場合には，基点レジスター９には，番地０からAまでのへだたりを入れておけばよい．レジスター９の内容と変位が自動加算されて得られた有効番地によって正しくデータを取り出すことができるであろう．

一方，ロード・ポイントにBが割り当てられた場合には，レジスター９には，番地０からBまでのへだたりを入れるとよい．レジスター９の内容と変位



第11図　基点レジスターの構想

でやはり正しくデータが取り出せるだろう.

これまで基点レジスターと呼んできたレジスターは特別な番地算出用のレジスターとして設けられているものではない. 普通に2進数演算に適用されている CPU 内に設けられた**汎用レジスター** (general register) を使うにすぎない. 汎用レジスターは機種によってその数は異なる. ある場合には8個であったり, 16個であったりする. この汎用レジスターのいくつかを基点レジスターとして利用するのである. アセンブラー言語でプログラムを作成するさいには, プログラムの中でどの汎用レジスターを基点レジスターに使うか宣言する必要がある. 「USING」というアセンブラー命令を使って基点レジスターを指定する. たとえば, つぎのような書き方をする.

```
USING     *, 9
```

第1オペランドは，どの位置を基点とするかを指定する．第2オペランドで基点レジスターを指示する．アセンブラーに基点が知らされるとそこを基準にして変位がどれだけかを算出してくれる．基点レジスターとして何が指定されており，変位がどれだけかがわかれば，命令で参照される記号番地を基点レジスターと変位の形式におきかえることができるだろう．なお，さきの USING の例で第1オペランドが＊印になっているのは基点をこの USING 命令のつぎの命令の最初の番地とすることを意味する．

アセンブラーがプログラムを翻訳する段階で，記号番地を基点レジスターと変位の形式に変換してくれたとして，実行の時点に基点になる番地を基点レジスターにロードすることは，どうやって行なえるだろう．実は，基点の番地をレジスターに入れる役目は，われわれのプログラムの中の命令で行う．各種の方法があるが，ここでその代表的なやり方を説明してみよう．**第12図**をみていただきたい．最初の命令「BALR　9，0」がこの役目を果している．

「BALR」（ブランチ連係命令）は，第1オペランド，第2オペランドとも汎用レジスターを指定する．この命令が実行されると，第1オペランドのレジスターに次の命令の最初の番地をロードしたうえで，第2オペランドのレジスターで指示されている番地にブランチする．もし，第2オペランドに0（ゼロ）が指定されていると，ブランチは生じないで第1オペランドで指定されたレジスターにつぎの命令の番地をロードするだけである．そもそも，この命令は，サブルーチンとの連係に使われる命令である．第1オペランドにサブルーチンからの戻り点を入れておいて，第2オペランドのレジスターで指示されたサブルーチンに飛び込む．

ここでは，第2オペランドを0にしているのでサブルーチンとの連係は意図していない．単に，汎用レジスター9に次の命令（つまり，MVC 命令）の最初の番地をロードしているにすぎない．ここで注意すべきは，USING 命令は何ら機械語にはならないことである．言語翻訳の段階でアセンブラーに基点の位置と基点レジスターに使用する汎用レジスターを指示するに過ぎない．したがって，BALR 命令の実行時にレジスター9に放り込まれる番地は，MVC 命令の最初の番地ということになる．この番地は，USING 命令で指示した基点

```
          BALR    9, 0
          USING   *, 9
          MVC     B, A        (6バイトの命令)
          AP      A, C        (6バイトの命令)
          B       NEXT        (4バイトの命令)
A         DC      PL5'0'      (5バイトの記憶域)
B         DC      PL5'0'      (5バイトの記憶域)
C         DC      PL3'0'      (3バイトの記憶域)
NEXT      〃
          〃
          〃
```

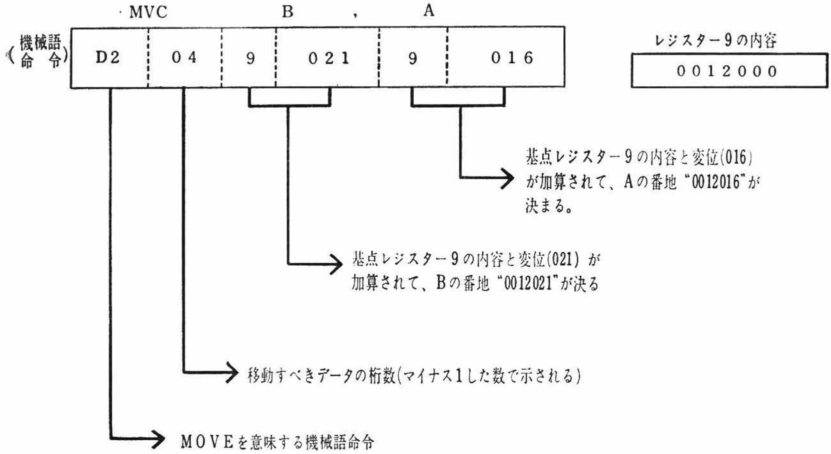
第12図　基点レジスターの指定と基点番地のローディング

の位置（＊印で指示）と全く一致することに注目していただきたい．

監視プログラムが，このプログラムを主記憶装置にロードしたあと実行が開始されると，BALR 命令によって，基点の位置を示す実際番地が基点レジスターにロードされることで，プログラムの再配置による番地の調整を行ったことになる．あとは，個々の命令が実行されるさい，基点レジスターの内容と変位がハードウェア的に自動加算されることで正しい番地を使った処理が行われるのである．なお，CPU で有効番地を算出する時間は，極めて高速であり全く無視してよいほどである．

ところで，アセンブラーが基点レジスターと変位を決定して機械語の命令を作る段階をもう少し詳しくみておこう．第12図の「MVC B，A」という命令を機械語に変換した結果が**第13図**である．データ記憶域を指示する記号B，Aともに基点レジスターと変位の形式からなる機械語に変換される．USING 命令でレジスター9を基点として宣言しているので，ともに9が基点レジスターとみなされる．

さて，変位はどのようにして算出されたのであろうか．第12図をふりかえってもらうことにしよう．基点は，MVC命令の最初の番地である．ここから記号B，Aまでのへだたりが変位である．アセンブラーは，個々の命令の長さ



第13図　機械語命令と基点レジスターの働き

（バイト数）は知っているし，個々のデータ記憶域の大きさはそれぞれ示されているので（L５とかL３）これを使って，記号番地までの変位を算出してくれる．Bの場合の変位は 6+6+4+5 で算出されるし，Aの変位は，6+6+4 で算出されよう．その結果Bの変位は21，Aの変位は16となるのである．

このMVC命令が実行段階でどのような番地となるかは第13図に示した通りである．BALR 命令でレジスター－9に 0012000 がロードされたと仮定して説明している．このプログラムが，異なった記憶域にロードされると，当然レジスター9にはそれに見合った基点の番地が放り込まれて，A，Bの正しい記憶域をさすように調整される．

なお，アセンブラー言語以外の言語，つまり COBOL，PL/I あるいは FORTRANなどでは，基点レジスターの選定や基点の設定を言語翻訳プログラムが自動的に行ってくれる．プログラマーがかかわりあいをもつ必要は全くないのである．

オペレーティング・システムにおいて，主記憶装置という資源を最大限に効率よく活用し，動的な管理を行ってゆくためには，プログラムの再配置を容易に行いうるようなハードウェア構造が要求される．このため，ハードウェアにおける番地指定方式の１つとして，基点レジスターの内容と変位による有効番

地算出という大胆な方法が採用されているのである．こうした構想が導入されることによって，はじめて本格的なプログラムの再配置が実現されることになり，柔軟性に富んだ多重プログラミングが可能となってきたのである．

八．入出力オペレーションと監視プログラムの働き

1．同時並行処理の重要性

入出力オペレーションと中央演算処理操作とを比較した場合，前者の処理速度がせいぜい100万分の１秒単位から1,000分の１秒単位であるのに，後者は10億分の１秒単位の処理を行う．当然，入出力オペレーション時間と中央演算処理時間の間に不均衡が生じてしまう，入力データが読み込まれないと CPU での処理の続行が一時中断される．あるいは，前に書き出し指令をだしたデータの出力が完了しないと，つぎのデータの書き出しができないという不合理が生じる．つまり，CPU の待ち時間が発生することとなる．この待ち時間を別の仕事のプログラムにふり向けることによって，多重プログラミングの状況を作り出すことは，すでにふれた通りである．

ところで，あるプログラムを実行している過程で，入出力オペレーションの完了を待たざるを得ない状態にあるとき，CPU を別のプログラムに活用できるための前提条件は何であろうか．もしも，目下の入出力オペレーションのために CPU が占有されてしまうような機構のハードウェアだとしたら，どうなるだろう．たとえ CPU に空き時間が生じているとしても，他のプログラムは実行出来ないことになる．つまり，多重プログラミングの前提として，入出力オペレーションを CPU の処理と完全に同時並行的に行いうるようなハードウェア設計が必要であるということになる．この入出力と CPU の並行作動は，必ずしも多重プログラミングを前提としなくても必要なことである．１つのプログラムにおいても，入力要請をだしたあとそのデータが読込まれるまでの時間を，当面の入力データを利用しない処理にふり向けることによって，全体としての処理時間を短縮したいという要望がでてくる．この場合でも入出力オペ

レーションとCPU処理のオーバラップが可能であることが前提になる.

そもそも，オペレーティング・システムは，コンピューティング・システムをはじめとした諸資源を最大限に活用することによって，システム全体の生産性の向上をもくろんでいるのである．そのためには，ハードウェアそのものが，オペレーティング・システムに協力しうるように設計されていることが必要である．入出力オペレーションに関して，ハードウェアに要求される第1のことは，CPUと最大限に並行操作が可能であるような構造になっていることである．もう1つの重要な要求は，複数個の入出力オペレーションが完全にオーバラップできることである．カード読取装置と印刷装置とが同時並行的に働いたら，あるいは，複数台の印刷装置が，他の諸装置と同時に作動することが可能ならば，システムの全体効率は一段と高まることになる.

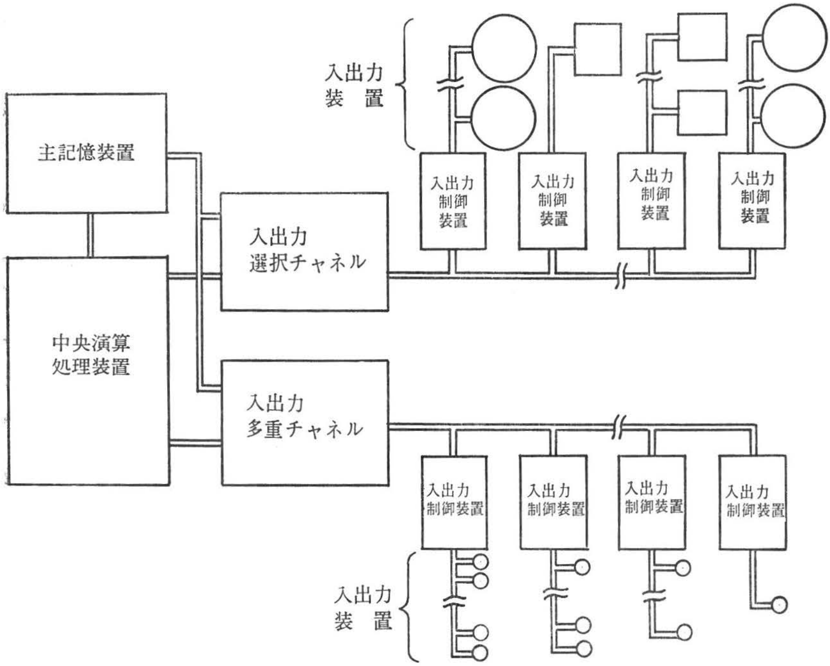
別の要請もでてくる．オペレーティング・システムは，あらゆる入出力装置を自己の配下において制御してゆくことを目指している．一方，入出力装置は，種々雑多である．様々な入出力装置を容易に統制するには，標準的手順で制御できるような工夫が，ハードウェア化されていることが望ましい．単に現有の入出力装置だけでなく，将来出現するであろう入出力装置をも制御できるような開放的工夫が機能化されていることが重要である.

このような配慮のもとで設計されたのが，入出力チャネルや制御装置である.

2.　入出力チャネル，制御装置，入出力装置の関係

入出力オペレーションの主体は，主記憶装置と入出力装置間での情報の転送である．入出力オペレーションには，**入出力チャネル**（input/output channel）**制御装置**（control unit）および入出力装置が関与する．もちろん，CPUも関与するが，出来るだけCPUが入出力オペレーションにわずらわされることを少なくすることによって，両者の同時並行作動が実現できることが望ましい.

CPU，入出力チャネル，入出力制御装置および入出力装置の相互関係を図示したのが**第14図**である．チャネルは，一方ではCPUと主記憶装置に接続しており，他方では入出力インターフェースを通して制御装置に接続している．**入**



第14図　入出力チャネル，制御装置の相互関係

出力インターフェース（I/O interface）とは，制御装置とチャネルを標準化した形式で接続する方式である．

インターフェースでは，各種の制御装置に共通的な情報形式と信号を与えることができるようになっている．したがって，あらゆる種類のチャネルと制御装置間のコントロールを標準化できる．このことは，新しい入出力装置が出現したとき，それを制御する装置を入出力インターフェースに標準的方式で接続することによって利用可能とする．いわゆる開放的設計を目指したものである．

チャネルの役目は，CPU が入出力装置と直接連絡するという仕事を代行することによって，CPU でのデータ処理と入出力オペレーションを同時並行的に行うことを実現する．それゆえ，入出力装置と主記憶装置間のデータ転送に

かかわる直接的制御は，チャネルの機能である．チャネルには，入出力オペレーションの制御に必要な共通的機能のすべてが賦与されている．これによって，CPU は，直接入出力装置とやりとりするという仕事から解放されることとなり，入出力オペレーションの最中にデータの処理を行うことが可能となる．

チャネルは，全ての入出力装置に共通的な機能，つまり読み込めとか書き出せといった機能の制御をする．しかし，印刷装置における用紙のスキップやスペース，磁気テープ装置における磁気テープの巻きもどしといった各装置固有の機能の制御は不可能である．こうした機能がチャネルから委譲されているのが制御装置である．このことから，制御装置は特定入出力装置を制御することを目的に設計されている．チャネルが，あらゆる入出力制御装置をその支配下におくのに対して，制御装置は特定の入出力装置を制御するのである．磁気テープ装置には，磁気テープ制御装置が，磁気ディスク装置には磁気ディスク制御装置が存在するゆえんである．制御装置は，チャネルと入出力装置の中継点に位置することから，一定のデータを貯えるバッファ装置をもっている．チャネルからビット単位で転送されてきた情報をバイト単位に集約して書き出すとか，逆に入力装置からバイト単位で送り込まれた情報を，チャネルにビット単位で直列転送するといったための緩衝記憶域である．印刷制御装置の場合には，チャネルを通して高速転送されてきたデータをまとめておき，低速で印刷装置に書き出すといった100バイトをこえるようなバッファを所有しているものもある．なお，ある種の装置では，制御装置が独立しておらず入出力装置にくみ込まれていることもある．

制御装置は，チャネルからの制御信号を受けとって，入出力装置固有の制御を行ったり，データ転送のタイミングの制御を行うだけでなく，入出力装置の状態をチャネルに連絡するといった信号手の役割りも引き受けているのである．

第14図をもう一度みていただきたい．1つの CPU および主記憶装置には複数個の入出力チャネルの接続が可能である．多くの場合，入出力多重チャネルは1台しか接続できないが，入出力選択チャネルは2台以上取付け可能であ

る．それぞれのチャネルは，全く独立して同時に作動することができる．それゆえ，CPU が，かなり高速のものであると，入出力オペレーションとのバランスを保った高速処理を行うために複数個のチャネルを接続するのが普通である．

1台の入出力チャネルには，複数個の異なったタイプの制御装置が取付け可能である．つまり，1台の入出力チャネルで異種の入出力装置を制御できることを意味する．このことは，さきにふれたチャネルの機能と入出力インターフェースの機能からも想像できたであろう．通常は，8台程度の制御装置を1台のチャネルに接続できるようになっている．

さらに，制御装置には複数個の入出力装置を接続できる．その接続台数は，制御装置の種類によってさまざまである．

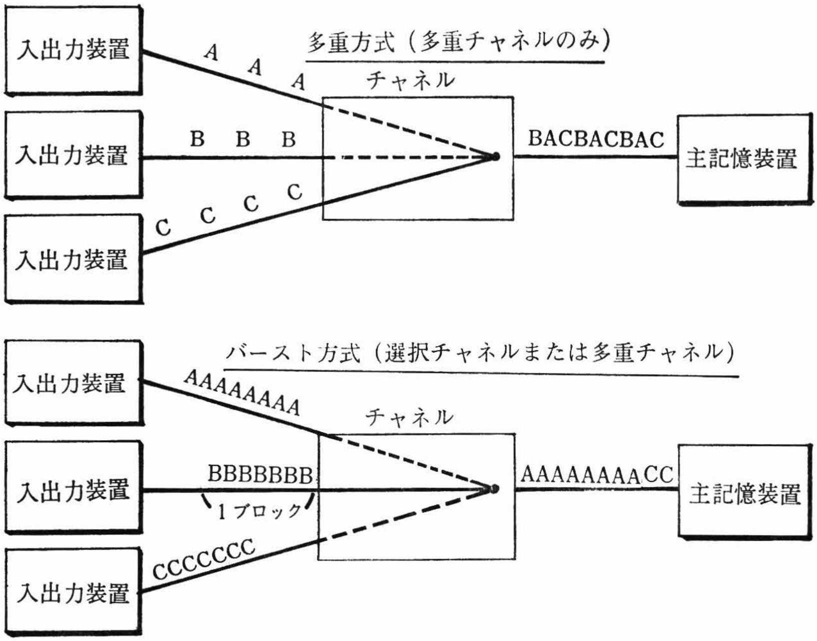
3. チャネルの作動方式

入出力チャネルには，**多重チャネル** (multiplexor channel) と**選択チャネル** (selector channel) の2種類があることは第14図に示した通りである．

選択チャネルは，磁気ディスク，磁気ドラムあるいは磁気テープといった高速の入出力装置のためのものである．一方，多重チャネルは，主としてカード，印刷，紙テープあるいはデータ通信などの低速入出力装置を制御するためのものである．ただし，多重チャネルで高速入出力装置を取り扱うこともできる．このことは，多重チャネルが，多重方式でもバースト方式でも作動することが可能であるということに由来している．これに反して，選択チャネルはバースト方式でしか作動しない．

チャネルの作動方式を図示したのが**第15図**である．チャネルの作動方式とは，主記憶装置と入出力装置との間でデータを転送する場合のやり方である．

多重方式 (multiplex mode) と呼ばれる作動方式の場合には，1つのチャネル内の機構が多数の入出力オペレーションによって共通的に使われる．この方式では，入出力オペレーションが一定の時間間隔に分割され，各時間間隔が特定の入出力装置と関係をもつことによって，その入出力装置のデータのうちの一部分だけが転送される．ある1つの時間間隔の間は，チャネルはただ1つの



第15図　チャネルの作動方式

入出力装置だけに接続されることになる．第15図では，3番目の入出力装置が接続されており，'C' という文字が転送されようとしている．この方式は，複数の入出力装置からの要求に応じてそれぞれの入出力オペレーションに対応する時間間隔が順番に与えられ，それぞれのデータ転送が入り組んで行われる．

いくつかの入出力オペレーションが並行的に入り組んで行われるためには，おのおのの入出力オペレーションの指示と制御が1つのチャネル内で行われていることになる．実は，多重チャネルは，いくつかの**サブチャネル**（subchannel）を有しており，それぞれのサブチャネルで同時的な入出力オペレーショを制御しているのである．各入出力オペレーションが1つずつのサブチャネルを利用することによって時分割処理を行っているのである．

このように多重方式で作動するチャネルでは，同時に複数個の入出力オペレーションが CPU の操作と並行して実行されてゆくのである．なお，このよう

な時分割が可能になるのは，各入出力オペレーションが比較的低速であることを前提とする．入出力装置から転送される1バイトあるいは複数バイトとつぎのバイトの転送の間に時間的余裕があるときのみ実現できることである．したがって，多重方式は低速入出力装置のオペレーションに対してのみ適用される．

他方，**バースト方式** (burst mode) では，ある特定の入出力装置に1ブロックのデータが転送されている間，あるいは逆に，1ブロックのデータがその入出力装置から主記憶装置に転送されている間中，チャネルが占有されたままである．したがって，このチャネルに接続されている他の入出力装置はこのブロックの転送が完了するまで，データを転送することができない．1ブロックを構成しているバイトとバイトの転送間隔に空きが生じない程高速であるため，その間に他の入出力装置のデータの入り込む余地が考えられないような高速入出力装置のオペレーションを担当する方式である．磁気ドラム装置，磁気ディスク装置あるいは磁気テープ装置などが，バースト方式でしかデータの転送ができないゆえんである．

前にふれたように選択チャネルは，バースト方式でのみ作動する．バースト方式では，どの瞬間をとってみても唯1つの入出力装置の制御しか行わない．同時点で2個以上の入出力装置を制御するという事態は生じない．したがって，選択チャネルは，1つのサブチャネルからなりたっている．多重チャネルは，多重方式で同時並行的に複数個の入出力装置を制御する必要があるのでいくつかのサブチャネルから構成されていた．うらをかえせば，選択チャネルは1つのサブチャネルしかもっていないので，多重方式でのオペレーションができないということになる．

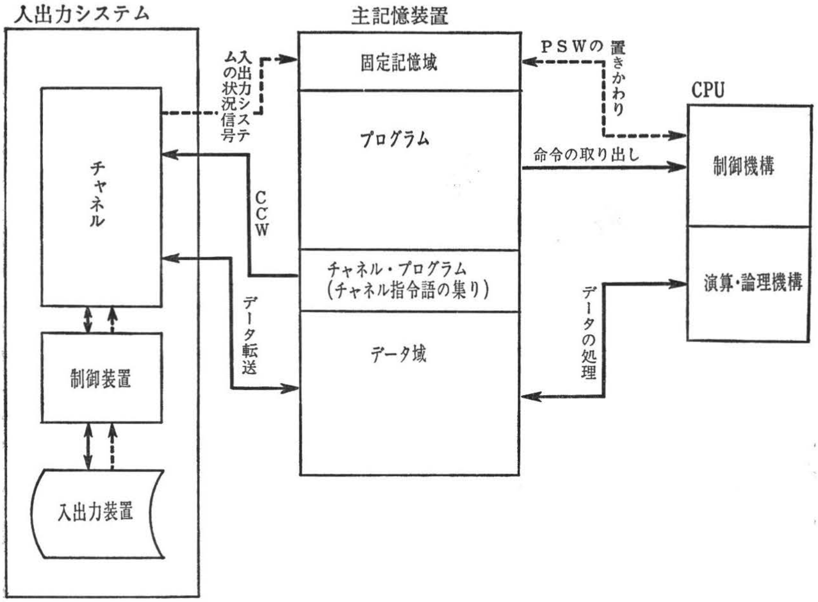
4. サブコンピュータとしてのチャネル

本来，コンピューターは，その作動の指示のすべてを CPU で行っていた．入出力オペレーションに関する一切の指示をも含めて CPU の役目であった．ところで，入出力オペレーションを完全に CPU のオペレーションとオーバラップさせ，かつ他方では複数個の入出力オペレーションをもオーバラップさせ

ようとすると CPU だけでは面倒をみきれなくなってしまう．1つの頭脳で複数個の入出力オペレーションの指示と監視を行うには限界がある．当然，その大半を CPU に代るべきものに代行させる必要がでてくる．実は，その代行者がチャネルなのである．チャネルは単にデータ転送の中継点にとどまっているのではなく，**入出力指令** (I/O command) を解読し，各装置に指令実行の指示を行い，その実行過程を制御するコンピューターである．チャネルがサブコンピューターと呼ばれるのはこのためである．もちろん，サブコンピューターといえども， CPU と完全に無関係に作動するわけではない． CPU の命令によって作動を開始するのであり，操作の完了は，入出力割込みによって CPU に伝達される． CPU のオペレーションとの同期化は十分に計られているのである．

第16図に示しているように， CPU で解析実行されるのは，普通のプログラムであるが，チャネルで実行されるプログラムは，**チャネル・プログラム** (channel program) と呼ばれるものである． CPU のプログラムは，四則演算命令，ブランチ命令，移動命令や回数制御命令あるいは入出力命令などから構成されているが，チャネル・プログラムは，読取り指令，書出し指令，逆方向読取り指令，あるいはチャネル・プログラムの実行順序を制御する指令など**チャネル指令語** (Channel Command Word 略して CCW) から構成されている． CPU プログラムからみれば，チャネル・プログラムは単なるデータの一種に過ぎない．決して， CPU で解析できるたぐいの命令ではない． あくまで，チャネルで解析され実行される．

チャネルや制御装置あるいは入出力装置からなる入出力システムのオペレーションの完了や異常事態が感知されると，その状況は固定記憶域にたくわえると同時に PSW の入れ代え，つまり，割込みが生じることで CPU に連絡される．この連絡がとどくまで， CPU は完全に入出力オペレーションと並行して，別のデータの処理をしてゆくことができるのである．入出力割込みが生じると，CPU では，固定記憶域にたくわえられた入出力システムの状況信号を分析することによって入出力完了の状況を把握する．特にエラー等がなく，しかも入出力待ち行列で待ちの状態におかれている入出力要請があれば，その開始

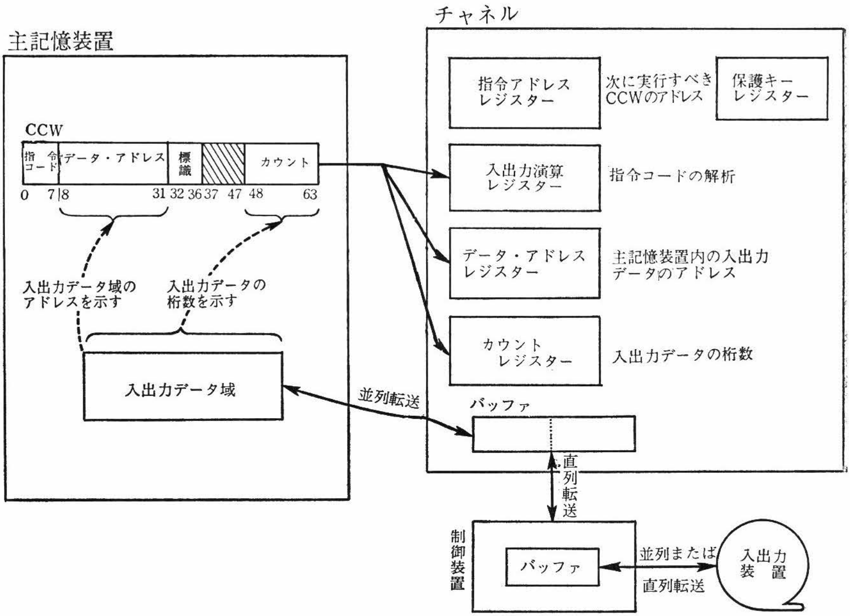


第16図　コンピューティング・システムにおける入出力システム

を促したあとで，入力データの処理やつぎの出力データの準備のルーチンにもどるであろう．特別なエラーがあれば，適切な処理ルーチンを実行することになろう．

サブコンピューターとしてのチャネルの働きをもう少し詳細にみることにしよう．その制御構造を図示したのが**第17図**である．チャネルで解析され実行が行われるのがチャネル・プログラムであることはまえにふれた．チャネル・プログラムは，入出力オペレーションの実質的な指示と制御を行うものであり，CCW からなりたっている．チャネル・プログラムは，ある場合には読み込めとか書き出せという１つの CCW からなり立っていることもあるし，一連の入出力指示を行うための複数個の CCW からなりたっていることもある．

まず，CCW の構成について述べよう．CCW は，64ビット，つまり８バイトからなる．第17図の左上に示してあるように，最初の８ビットで指令コードを表わす．指令コードの種類としては，つぎのようなものがある．



第17図　サブコンピューターの役割りを果すチャネル構造

① 読取り（read）：入力装置からレコードを読込むための指令.

② 書出し（write）：出力装置へのレコードの書出しのための指令.

③ 逆方向読取り（read backward）：磁気テープ装置に対してのみ有効な指令であり，磁気テープを逆方向から読込むための指令

④ 感知（sense）：該当する入出力装置で行なわれた最新の入出力オペレーション中に発見された異常な状態と入出力装置の各種状況を主記憶装置に送り込むための指令.

⑤ チャネル内とび越し（transfer in channel）：チャネル・プログラム内でブランチを指示する指令.

⑥ 制御（control）：入出力装置に対して，その入出力装置特有の働きを行わせる指令．たとえば，磁気テープの巻きもどしとかバックスペースを行わせるといった働きのものであり，実際にはチャネルで実行されるのではなく，制御装置や入

出力装置で解析され実行される．

多くの入出力オペレーションにおいては，データの転送を伴う．したがって，主記憶装置のどの番地から出力するとか，どの番地に読込むという指示がCCW の中に示されていなければならない．これを示すのが，CCW のビット8～31の位置のデータ・アドレスである．何桁のデータの読み書きかは，ビット位置48から63のカウントで示される．

CPU がチャネルに対して入出力オペレーションの開始を命ずる（SIO 命令を使う）と，主記憶装置にあるCCWがチャネルに内蔵されているレジスターにロードされる．CCW を主記憶装置のどこから取り出してくるかは，**指令アドレス・レジスター**によって分る．CCW の指令コードは，**入出力演算レジスター**に登録され，かつ解析される．入出力域のアドレスが入っているデータ・アドレス部は，**データ・アドレス・レジスター**にロードされる．また，入出力データの桁数を示すカウントは，**カウント・レジスター**にロードされる．ちょうど，CPU を構成するレジスターで命令が解析されてデータ処理が行われると同じように，チャネルでもCCW が各種レジスターで解析されて実行されるのである．CPU の PSW に保護キーがセットされて，データの取出しや記憶に関する保護が行われると同じように，チャネルにも保護キーが送り込まれ，目下のプログラム処理に関係する記憶域以外の場所に入力したり，逆に出力することのないように監視が行われる．

主記憶装置とチャネルのバッファの間でのデータ転送は，1バイト単位あるいは複数バイト単位で並列に転送される．一方，チャネルから制御装置への転送は，ビット単位である．したがって，入力されるデータは，チャネルのバッファでバイト単位に組み立てられたあとで，主記憶装置に並列転送される．入出力装置と制御装置の間では，ある装置では並列転送されるものもあるし，別の装置では，ビット単位で直列転送されるものもある．

これで一応，サブコンピューターとしてのチャネルの働きがつかめたことと思う．なお，多重チャネルが複数個のサブチャネルから構成され，同時に複数個の低速入出力装置のオペレーション制御，つまり，多重方式のオペレーションが制御できるのは，各サブチャネルごとに前述のような各種レジスターを有

しているからである．それぞれのサブチャネルが指定されたチャネル・プログラムを並行的に実行しているのである．

5. 入出力オペレーションの流れと入出力監視プログラム

これまで，チャネルを中心に入出力オペレーションをながめてきたのであるが，CPU オペレーションと，どのようなかかわりをもって展開されるのかは深く追及しなかった．ここで，入出力オペレーションが CPU プログラムとの関係でどのように処理されてゆくかを検討してみよう．

そもそも，入出力オペレーションの要請は，われわれプログラマーが作成したプログラム，つまり，問題処理プログラムで行う．GET，PUT 命令あるいは READ，WRITE 命令のたぐいがそれである．それに，ファイルやレコードの形式の定義もわれわれのプログラムの中で行う．アセンブラー言語の場合には，DCB とか DTF とかの定義命令があるし，PL/I 言語では，DECLARE がそれに当る．COBOL では，DATA DIVISION があるし，FORTRAN では FORMAT ステートメントなどで定義する．これら定義命令でレコードの桁数とか入出力域が指示されるので，これらの情報を使って自動的にチャネル・プログラムが作り出されるわけだ．特に，われわれがチャネル・プログラムを詳しく記述する必要がないのは，このためである．

ところで，入出力命令である GET，PUT あるいは READ，WRITE 命令などはどのような働きをするのであろうか．実は，これらの命令は入出力割込みを要求する命令にすぎない．処理プログラムで能動的に要求する割込みである監視プログラム呼び出し命令（SVC）におき代るものである．**第18図**の①で示みしたのがこれである．当然のことながら，割込みが生じたことによって，PSWが入れ換る．監視プログラム呼び出しの新 PSW が CPU の PSW になる．この新 PSW の命令アドレスには，監視プログラム呼び出し割込み処理ルーチンが示されているので，このルーチンから実行が開始される．このルーチンでは，SVC 命令のオペランドで指示された割込みコードを分析したうえで，入出力監視プログラム部分にブランチする．結果としては，第18図の①で示したように制御プログラムの一部を構成する入出力監視プログラムが実行されるこ

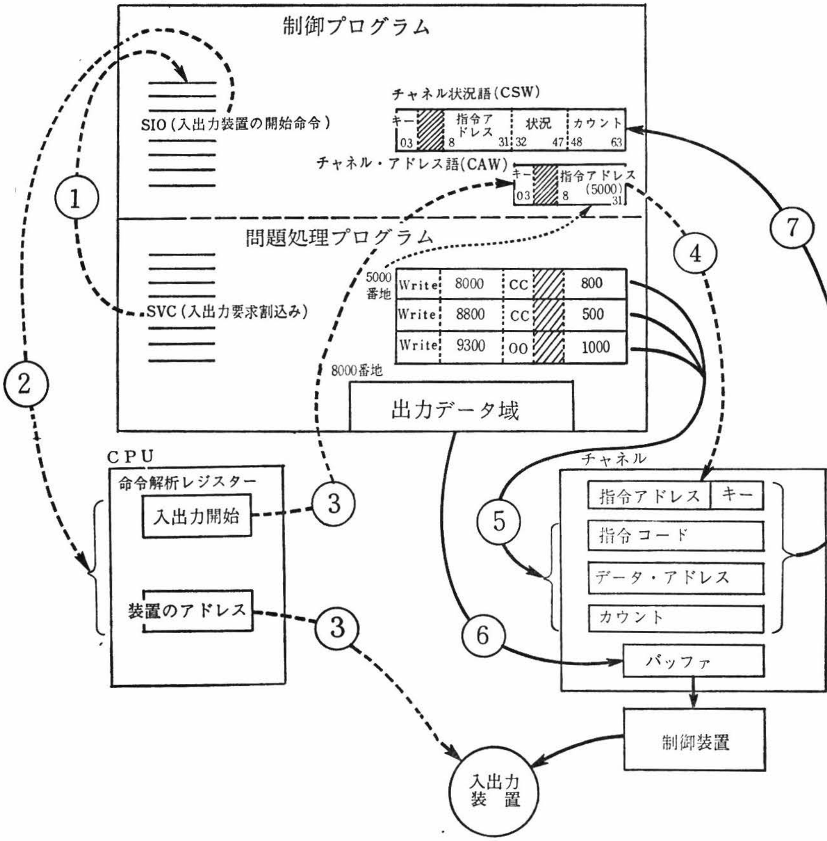
となる.

入出力監視プログラムの中に入出力オペレーションの開始を要求する SIO 命令がある. この命令は監視プログラム状態においてのみ実行可能な命令, すなわち, **特権命令**と呼ばれるものである. 特権命令である以上, 問題処理プログラムで使用することは出来ない. 監視プログラムのために設計された特別な命令である.

この SIO 命令のオペランドでは, 入出力オペレーションの開始を要求するチャネル, サブチャネルおよび入出力装置が指示される. この命令が CPU で解析されると, 指示されたチャネル（あるいはサブチャネル）と入出力装置が使用可能状態にあるかどうか調べられる. 入出力開始命令（SIO）が実行されている過程で, エラーや例外状態が検出されるとか, チャネルやサブチャネルが目下他のオペレーションで使用中であるときには, その旨の信号が返され入出力オペレーションは開始されない（第18図②および③参照).

指定されたチャネル, 入出力装置が使用可能の状態にある場合には, 主記憶装置の72番地から始まる4バイトから構成される**チャネル・アドレス語**(CAW : Channel Address Word) がチャネルに送り込まれる. チャネル・アドレス語の最初4ビットには保護キーが貯えられており, これがチャネルのキー・レジスターに入る. この保護キーは, PSW に貯えられている保護キーと同様の意味をもっている. つまり, 入力オペレーションにおいては, この保護キーに一致する記憶域以外に読み込もうとしたとき割込みが生じる. 逆に, 出力にさいしては, この保護キーに一致する記憶域以外から書き出すという違反をおかしたとき割込みが生じるわけだ. つぎの4ビットは, いつも0ビットである. 残り24ビットでチャネル・プログラムのうちの最初に実行すべき CCW のアドレスを示す. これが指令アドレス・レジスターにロードされる. チャネルにおいては, CCW が1つ実行されるごとにこの指令アドレスが更新され, つぎに実行すべき CCW の場所を示す（第18図④参照).

CAW の情報がチャネルのそれぞれのレジスターに登録されると, あとはチャネルの主導権でチャネル・プログラムが実行されてゆく. CCW の指令情報が, 入出力演算レジスター, データ・アドレス・レジスター, カウント・レジ



第18図　入出力オペレーションの順序

スターのそれぞれにロードされ解析・実行される．CCW の標識の部分に CC と記入したのは，その CCW が実行されると自動的につぎの CCW がチャネルで解析され実行されることを意味している．図中の CCW の指令コードは Write（書き出し）と仮定しているので，主記憶装置の出力データ域の連続する 2300 バイトの内容が3つのレコード（最初は 800 バイト，つぎが 500 バイト，3番目が 1000 バイト）として書き出される（第18図⑤と⑥参照）．

チャネル・プログラムの実行中にエラーや異常事態が検出されたり，あるいは正常にチャネル・プログラムの実行が完了すると，主記憶装置の64番地から

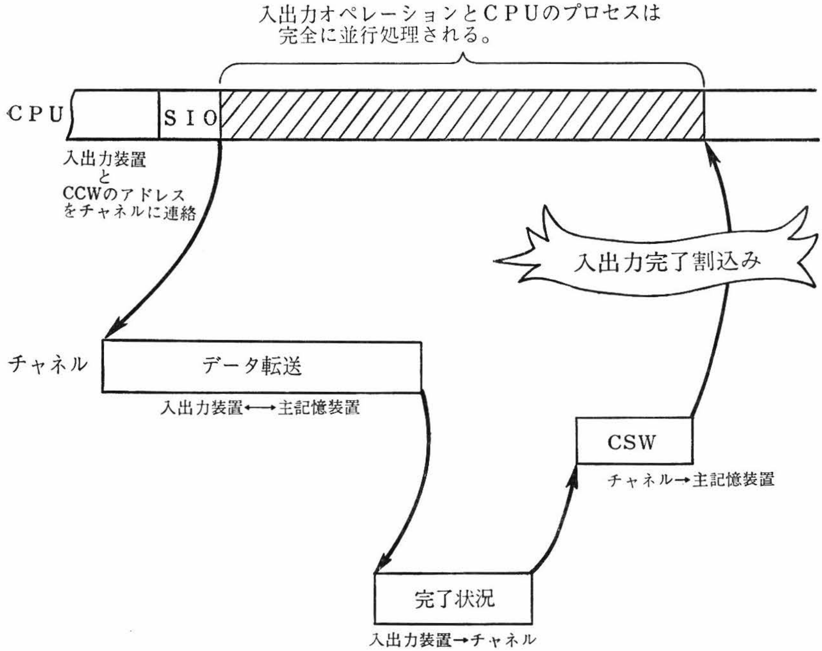
はじまる64ビットに入出力オペレーションの終了に関する情報を送り込むと同時に入出力割込みが生じる．この割込みによって，CPU の PSW は，入出力割込みの新 PSW とおき代る．つまり，入出力監視プログラムに再びコントロールがもどされることになる．

64番地からはじまる64ビットの記憶域は**チャネル状況語**（CSW：Channel Status Word）と呼ばれる．チャネル状況語は，入出力オペレーションが終了したときの状況を入出力監視プログラムに伝達する働きをしているのである．入出力オペレーションが終了したときの保護キーの内容，どの CCW を実行して終了したか，どのような理由で入出力が終了したのか（無事入出力オペレーションを完了したのか，それとも何らかのエラーや異常事態が発生して終了したのか），および CCW のカウント部で示された入出力桁数のうち，何桁を残して終了したのかなどが示される．入出力割込みによって入出力監視プログラムに制御権が渡ると，このプログラムでは，CSW の内容を分析することによって，エラーや異常事態の原因をつかみ，それ相当の処理を行う．正常に完了した場合には，つぎの処理に移ることになる（第18図⑦参照）．

第19図にまとめたように，CPU とチャネルの並行操作は，つぎのようになる．入出力開始命令（SIO）が実行されることによって，両者の並行操作は開始される．ここから，チャネルは独力でその機能を果すようになるのである．CPU では，別のプログラムないしは，目下の入出力オペレーションに関係するプログラムのうちで入出力データを必要としない処理が並行的に行われる．チャネルでは，CCW が解読され，データ転送が行われる．入出力が完了すると完了した時点の入出力装置の状況がチャネルに送り込まれ，チャネルから主記憶装置の CSW に貯えられると同時に割込みが生じる．この割込みが生じるまでの時間，チャネルのオペレーションは完全に CPU のオペレーションとオーバラップしているのである．

6. 入出力監視プログラムの概念

入出力オペレーションのための監視プログラムには，きわめて複雑な機能を果すことを要求される．SIO 命令を発するまでの諸準備，CAW の設定や CCW



第19図　CPU とチャネルの並行操作

の修正などから始まり，SIO 命令を出したあとでのチャネルや入出力装置の作動状況の分析が必要である．いったん，チャネルに入出力制御権が渡ったあと，入出力オペレーションが終了すると割込みが生じる．この割込み処理も当然ながら，監視プログラムの守備範囲である．終了の理由を分析し，それぞれに見合った適正措置を行うことが必要である．

ところで，多重チャネルの場合だと，1つのチャネルで同時並行的に複数個の入出力オペレーションが競合する．チャネルが複数個あれば，チャネル同志で同時並行操作が行われよう．それぞれの入出力オペレーションに対して入出力開始命令が実行され，入出力完了に伴って割込み処理が生じる．1つの入出力オペレーションだけをとりあげても前述のように複雑な処理を必要とするのである．ましてや，複数個の入出力オペレーションが錯綜してくるとお手上げになってしまうだろう．ハードウェアの本来的機能としてチャネルと CPU の

同時並行操作が組み込まれていたとしても，その機能をソフトウェアで 100 パーセント活用できないとしたら，その意義は激減してしまうことになりかねない．

だが，あまり深刻になる必要はない．この複雑な入出力オペレーションは，プログラマー泣かせではないのだ．そもそも，入出力オペレーションがこみ入ってしまったのは，コンピューティング・システムのシステムとしての全体的効率を高めることに着眼した結果に他ならない． CPU とチャネルが，チャネル同志が，あるいはいくつかの入出力装置が最大限並行操作を行うようにするには，そう簡単な制御で間に合うはずはないのである．当然，複雑なハードウェア機構とならざるを得ないし，それを監視，制御するソフトウェアもむずかしくなってしまう．ところで，このような入出力オペレーションの処理を問題処理プログラムに組み込むという思想は，ハードウェア設計の段階から，除外されていたのである．とても，問題処理プログラムでカバーしきれるものではないということもさることながら，このハードウェア機能を充分に生かしてゆくためには，統一的立場からの標準化された制御が必須だからである．それぞれのプログラマーが自分の意図通りの入出力処理ルーチンを作り上げたのでは，並行処理効率も落ちるし，誤りをおかす危険性もある．それに，いくつかの同時に発生する入出力オペレーションをうまく処理してゆくためには，どうしても中央集権化された監視が必要となるのである．

ということから，入出力オペレーションのうちで，ハードウェアとかかわりをもつ部分の制御のすべてが，入出力監視プログラムにゆだねられることになる．ハードウェアとかかわりをもつ部分とことわったのは，たとえば，ブロック化されたレコードを論理レコード単位にばらしてゆくといった直接入出力ハードウェアとかかわりをもたない部分に関しては，入出力監視プログラムの機能外だということを意味する．

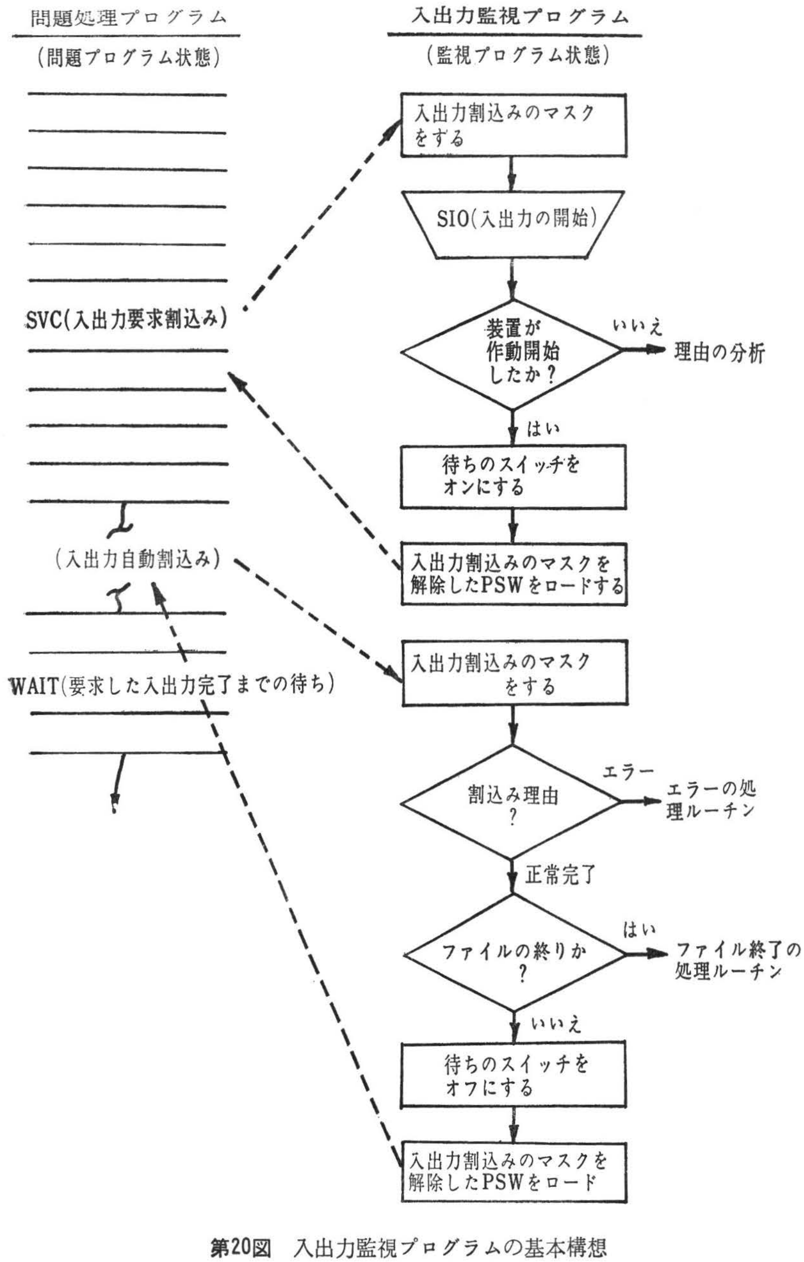
ここで，簡単に**入出力監視プログラム**（input/output supervisor）の機能構造をまとめておこう．**第20図**にその概略を流れ図として示した．入出力監視プログラムは，制御プログラムの一部分であり監視プログラム状態で働く．すでに何回もふれたように問題処理プログラムで，何らかの入出力関係の要請命令

が出されると，監視プログラム呼び出し（SVC）命令が実行され割込みが生じる．割込みルーチンで，入出力関係の要請であることが判別されると，CPUの制御権は，入出力監視プログラムに渡る．この入出力監視プログラムでは，まず，入出力割込みのマスクをする．つまり，PSW のビット位置0から7のシステム・マスク部分をゼロ・ビットの状態にしておく．これによって，入出力監視プログラムの実行中の入出力割込みを一時延期する．目下の入出力要請の開始が終了するまで，入出力割込みが生じても待たせておくことによって，混乱を防ぐのである．

入出力開始の諸準備を終えたあとで，SIO 命令を実行する．首尾よく，該当入出力装置の作動が開始できたときには，つぎに進む．該当チャネルや入出力装置が他のオペレーションに占有されているとか，エラーや異常事態が発見されたときには，それに見合った処理手順がとられる．入出力装置の作動開始が確認されると，待ちのスイッチをオンにする．目下の入出力オペレーションが完了するまで，それに関連する処理はできないという目印を特定記憶ビットに貯えることを意味する．たとえば，入力要請の場合，待ちのスイッチがオフであることが確認されるまでレコードの入力が完了していないことになる．したがって，オフが確認されるまで，そのレコードに関係する処理命令は出せないのである．一般には，WAIT 命令でこのスイッチの状態がテストされ，スイッチがオフになるまでは，つぎの命令の実行に入らない．

待ちのスイッチをオンにしたあとで，入出力割込みの延期を解除した PSW をロードする．もし，何らかの入出力割込みが延期されていると，その割込みがただちに起ることになる．延期されている入出力割込みが存在しない場合には，SVC 命令のつぎの命令から実行される．なぜなら，ロードする PSW は，さきの SVC 命令による割込みで貯えられた旧 PSW だからである．延期されている入出力割込みがとられたとしても，終局的には，SVC 命令のつぎにもどってくる．

入出力監視プログラムのもう１つの重要な部分は，**入出力割込み処理ルーチン**（I/O interrupt handler）である．入出力割込みは，入出力オペレーションにおいてエラーをも含めた異常事態が検出されたとき，あるいは正常にオペレ



第20図　入出力監視プログラムの基本構想

ーションが完了したときのいずれでも生じる．

このルーチンの基本構造は，入出力開始ルーチンの場合と同様である．まず，入出力割込みのマスクをする．その理由は入出力開始の場合と同じである．そのあとで割込みの理由を分析する．分析のもとになる情報は，CSW から得られる．エラーによる割込みだと，そのエラーの理由を詳細に分析し，それに見合った訂正措置の手順がとられる．割込み理由が，入出力の正常な完了を意味する場合には，入出力ファイルの終りまで処理された状態かどうかチェックする．もし，ファイル全体を処理し終っているのであれば，ファイル終了の後始末の処理手順がとられる．残余のレコードがあるとか，まだ書き出しのスペースがあるならば，つぎのステップに進む．入出力オペレーションは正常に完了しているのだから，入出力開始ルーチンでオンにしておいたスイッチをオフにする．当該入出力装置は，使用可能状態であることを意味すると同時に，入力オペレーションの場合には，入力レコードの処理が可能であることを意味する．入出力割込みの延期を解除する PSW をロードすることによって，入出力割込みが生じた次の命令の実行に移る．もちろん，延期されている入出力割込みが存在していればその割込み処理が完了したうえで，元の入出力割込みのつぎの命令にもどることになる．

このような構想にもとづいた入出力監視プログラムが入り組んだ入出力オペレーションのサービスを行ってくれるのである．

第 4 章

プログラム設計構造と連係編集プログラム

一．は じ め に

コンピューターの急速な普及とその利用分野の劇的な拡大傾向は，プログラム作成に関係した重要な問題を投げかけている．まず第1に，プログラム設計開発にたづさわる人間，つまり，プログラマーが慢性的枯渇化している状況を作り出している．一方，新たなコンピューターの適用業務の数と分野は，とどまることなく拡大しつづけているのが現状である．このことは，プログラマーの量的増大の要請だけでなく，新たな分野の知識や手法に耐えうる能力を有した人間の渇望へと発展してきている．

とはいうものの，質的に優れた人材を確保し，あるいは早期養成を計ることは至難の状況でもある．その結果，コンピューターの効果的利用がある程度犠牲にされるとか，あるいは重要な適用業務へのコンピューター利用の計画がそこなわれがちである．

このような問題に直面していることに対して，オペレーティング・システムはプログラム作成のための各種便宜を計ってくれる．新しいプログラム作成に要する時間，労力あるいは費用を軽減するための工夫がこらされているのである．PL/I, COBOL, ALGOL, FORTRAN あるいは RPG といった各種言語翻訳プログラムや連係編集プログラムとかディバッキング・プログラムといったサービス・プログラムの機能がオペレーティング・システムにくみ込まれていることである．

ところで，これらプログラムが，それ自身重要な機能をもっていることは，今更いうまでもないことであるが，実は，これらプログラムの機能が統合化されて利用できるように設計されていることにこそ，オペレーティング・システムの意義がある．

つまり，制御プログラムの機能と結合されてこそ，プログラムの作成や保守に対して有効な力を発揮してくれるのである．統一され標準化された制御システムであるからこそ，新しいプログラムの開発や保守に要する時間や労力が節約できるといってよかろう．オペレーティング・システムは，複雑で高度な適用業務のプログラム作成にさいしては，何人かのプログラマーが協同して1つのプログラムを分割作成することがきわめて普通に行えるようにしている．

プログラムの修正や更新にさいしては，関係するモジュールにだけ手を加えて容易にもとのプログラムに組み入れることができるようにしている．さらに，一たん作成した共通的サブルーチンは，どのプログラムからでも協同利用ができるようにすることで，プログラム作成に要する労力や時間の節減が計れるようにしている．

この種のプログラム作成や利用上の援助を，オペレーティング・システムは次のような統一された方法で行おうとしている．まず，プログラムの構成や設計にさいして，**モジュール**（module）という概念を導入することを主眼としている．いわゆる“**積木方式**”のプログラム構成手法である．1つのプログラム設計にさいして，プログラムを複数個の論理的単位に分割してプログラム構造を作り上げ，各論理的単位（モジュール）を別々に作成し，結合する方式である．この積木方式の構造は，オペレーティング・システムの制御プログラムを設計するさいの基本構想でもあるが，この考えがわれわれが作成する問題処理プログラムにもそのまま通用するように配慮されている．

つぎに，各種プログラムの格納にさいしては，直接アクセス記憶装置を利用した組織的プログラム・ライブラリーを提供し，かつ統一的なライブラリー管理を行ってくれる．これによって，プログラム・ライブラリーから所用のプログラムが自動検索され，すぐに取り出せるようになるし，ライブラリーに登録されている既存のプログラムの修正・変更といったプログラム保守作業も簡単

に行えるようになる．それに，オペレーティング・システムでは，仕事（タスク）の実行中に必要なプログラムやサブプログラムをダイナミックに主記憶装置にローディングしてくれる機能もある．この機能があるので，複雑でかつ大きなプログラムに関して，あらかじめプログラム・オーバレイの構造を作り上げておく必要をなくすることもできる．

この章では，積木方式のプログラム構造，プログラム・ライブラリーの構造，動的プログラムの設計および連係編集プログラムの機能と使い方などについて説明してゆこう．

二．積木方式のプログラム構造とプログラムの結合

そもそも，積木方式（モジュラー）の概念は，コンピューターのソフトウェアというよりも，むしろハードウェアに採用された基本設計構想であった．ハードウェアの技術革新がめざましく，かつその利用技術が拡大するにつれて，コンピュータ・メーカとしては，変化に応じることができ，広い応用分野に対応できて，かつ古いシステムから新しいシステムの互換性のある融通性の富んだハードウェア設計の必要性に早くから直面していた．

初期のコンピュータ・ハードウェアが硬直した剛構造であり，新しい技術や応用分野が出現するとたちまちのうちに通用しなくなったのに対して，第2世代から第3世代のコンピューターが変化に応じるためのダイナミックな柔構造で設計されることによって，システムの互換性を保ち，かつ，新しい入出力装置やその他の技術をそのハードウェア・システムに容易に組み入れることができるようになったし，内部回路などに関しては量産化を計ることもできるようになったのである．

モジュラー構想は，単にコンピューターの世界だけでなくあらゆる分野において，変化に応じることのできるダイナミックな柔構造として注目され，応用されてきている．1964年の東京オリンピックの競技場の設計や最近の高層建築設計，新しい都市計画や住宅設計などに少なからず影響を与えているし，エキスポ'70の各種建築物や芸術にも反映されている．最近では，企業経営におけ

るマネージメントや教育におけるコース体系などにも，このモジュラー構想が採用されているのである．

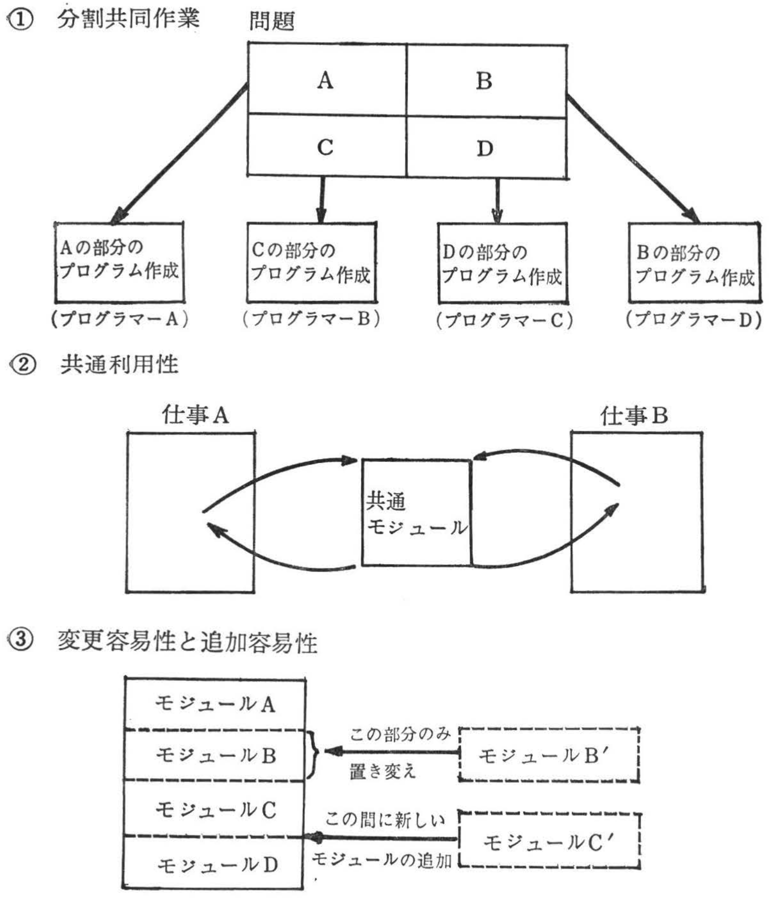
1.　積木方式のプログラム構造の意義

1人のプログラマーが，プログラムを書き上げ，言語翻訳を行って，すぐに本番の仕事の処理に移ることは，可能ではあるが，一般的には極めて例外的なことである．大半のプログラムは，テストをしたうえでデバックし，修正を加えるといった手順をくりかえしたうえで，はじめて，本番の仕事に使えるようになる．1つのプログラムが設計開発されて完全になるまでに数ケ月も要することは珍らしくはない．そのうえ，一たん完成したプログラムでも，性能の向上だとか処理方式の更新あるいは機能拡大のために絶えず手が加えられるのが実情でもある．

このような状態を考慮すると，プログラムの設計や開発，保守といった作業を容易にかつ正確で効果的に行いうる手法の要請がでてくる．それが，プログラムの積木方式である．プログラムやサブプログラムをモジュール形式で設計するのに適合するように言語翻訳プログラム，連係編集プログラムや制御プログラムが，統一的目標のもとでデザインされていることによって，各モジュールを別々に作成したうえで，1つのプログラムに結合して使用できるようになる．

第1図の①で示したように，与えられた1つの大きな問題を，プログラムの設計段階でいくつかのモジュールに分割する．各モジュールは，入力関係だとか，レコードの更新処理関係あるいは出力関係などと一定の論理の集まりで分割する．分割したモジュールを1人のプログラマーが担当することもできるが，普通何人かのプログラマーが別々に担当する．分割共同作業によって，プログラム作成に要する時間は短縮されるし，各モジュールごとに分離テストを行ったうえで全体を結合すればよいので，デバッキングは容易になるし正確なテストが行える．テストのつど，全体を翻訳し直す必要はなく，不完全なモジュールだけを取りだして翻訳チェックするだけでよいからだ．

そのうえ，各モジュールは，処理内容と形態に一番ふさわしいプログラム言



第1図　積木（モジュラー）方式の意義

語を採用してプログラミングができる．たとえば，数理的手法を駆使する必要のあるプログラム・モジュールでは FORTRAN 言語か PL/I 言語を適用すればよいし， FORTRAN 言語や PL/I 言語では取扱いが困難であるような特別な処理だとか監視プログラムの機能を利用したいモジュールではアセンブラー言語を使えばよい．まさに，“選択の時代”にふさわしいアプローチができるのである．

第1図の②　に図示したようにモジュラー形式にしておくと，各種の仕事で

共通的に必要となる処理部分を抽出して1回作成しておくと，異った仕事のプログラムで共同利用が可能となる．共通モジュールは，言語翻訳の段階で主プログラムに組み入れることもできるし，連係編集の段階あるいは主プログラムの実行中に組み込むこともできるようになっている．

さらに，③で示しているように変更や追加が容易に行える．モジュラー構想を採用しないプログラム設計では，プログラムのどこか一部に誤りがあったり，変更の必要性が生じたときには，手直しをしたうえでプログラム全体の再翻訳からやり直す必要がある．それに，モジュラー構想のように一定の論理的関連性のある部分を体系的に一まとめにすることを行っていないので，手直しを必要とする部分を見つけだすことも容易ではない．ところがモジュラー方式だと，手直しをする部分が容易につかめるだけでなく，それに関係するモジュールだけを再翻訳して古いモジュールと置きかえるだけでこと足りる．

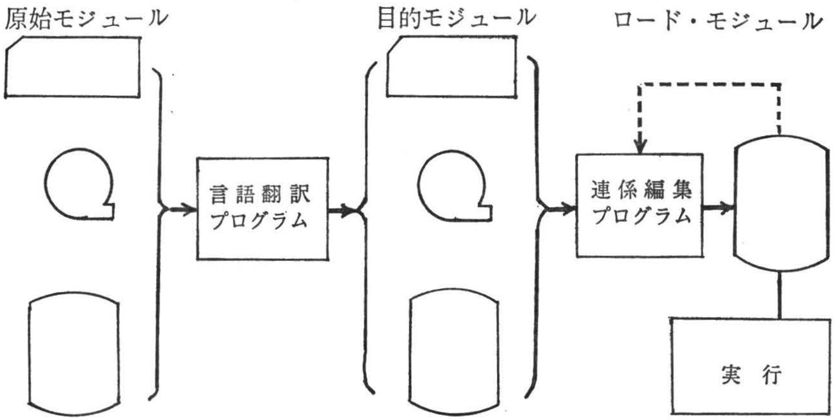
このことは，プログラムの保守が再三，再四生ずるだけに馬鹿にならない．不測の誤りが生じたことによる手直し，社内の諸制度の変更や国の法律の変更による手直し，プログラムの中で使用している手法の更新による手直し，あるいは処理効率を上げるための手直しなどけっこう多いのである．さらに，現有プログラムに新しい機能を追加する要請もでてくる．プログラムが複雑で大きなものだと，機能拡大のためにプログラム全体を書き直すことは不可能に近い．ところが，モジュラー設計になったプログラムでは，新しい機能を含んだモジュールを作成し，現有プログラムの一部に追加すればよい．もちろん，この新しいモジュールへの制御権やデータの受け渡しのために関連モジュールの一部手直しの必要が生じることもある．

このようにモジュラー方式のプログラムは，変化に耐えうる柔軟性に富んだ設計構造であり，したがって，長期的寿命を保ちうるプログラム構造でもある．

2. プログラム・モジュールの種類

プログラム・モジュール（program module）を正確に定義するとつぎのようになる．他のプログラム部分に影響を与えることなく修正や置きかえができ

るプログラム部分または完全なプログラムのことである．したがって，プログラム・モジュールの特別な形態としては，１つのモジュールが完全なプログラムを形づくる場合もある．つまり，分割されないプログラムである．一般には，プログラム・モジュールのことを，サブプログラムと呼んでいるが，サブプログラムという用語は，特に言語翻訳プログラムで出力された単位として使われることもある．



第2図　モジュールの形態

プログラム・モジュールには３種類のタイプがある．**原始モジュール**（source module），**目的モジュール**（object module）および**ロード・モジュール**（load module）である．**第2図**に示しているように，これらのタイプは，プログラムの作成過程における段階を表わしているに過ぎない．

原始モジュールは，言語翻訳プログラムへの入力単位となるものである．言うまでもなく，プログラマーが書いた一連のステートメントからなりたっている．穿孔カードに記憶されている場合もあるし，磁気テープ上に収容されている場合もある．

さらに，直接アクセス装置上にもうけられた**原始モジュール・ライブラリー**（source module library）に登録されている場合もある．

目的モジュールは，言語翻訳プログラムの出力結果であり，機械語のモジュールである．しかし，このままでは実行できない．連係編集プログラムで処理

されることによって，実行可能なモジュールとなる．目的モジュールは，プログラムの一部分であることもあるし，完全な1つのプログラムを構成していることもある．原始モジュールと同様，穿孔カード，磁気テープあるいは直接アクセス装置上の**目的モジュール・ライブラリー**（object module library）に収容される．

ロード・モジュールは，連係編集プログラムによって作り出される．ロード・モジュールは，1つ以上の目的モジュールから作られる場合もあるし，目的モジュールとロード・モジュールを連係編集プログラムに入力することによって，両者が結合したロード・モジュールとなる場合もある．いづれにしろ，ロード・モジュールは，直接アクセス装置上の**ロード・モジュール・ライブラリー**（load module library）に収容される．

原始モジュールは，別の関連モジュールで使われている命令やデータを参照する記号をもつことができる．また，逆に別のプログラムから参照される命令やデータをそのモジュールに包含している場合もある．しかし各原始モジュールは別々に言語翻訳されるので，この段階では他のモジュールと結合されない場合がある．したがって，言語翻訳の過程では，他のモジュールを参照したり，逆に他のモジュールから参照されたりする記号に関する情報を**外部記号ディクショナリー**（External Symbol Dictionary ：略して ESD）に集約して，目的モジュールの一部として出力する．

この情報を使うことによって，連係編集プログラムは，複数個の目的モジュールを結びつけてくれる．連係編集プログラムは，言語翻訳の段階で解決できなかった外部参照記号について，外部記号ディクショナリーをつき合わせることによって，その記号が参照している項目の番地に置きかえてくれるのである．

連係編集プログラムは，2つ以上のモジュールを結合する働きがあるが，出力されたロード・モジュールにもそれぞれの目的モジュールに存在した外部記号ディクショナリーの内容を整理統合した新しいディクショナリーを組込んでくれる．

この新しいディクショナリーには，すでに解決ずみの外部記号と，まだ未解

決の外部記号が入っている. 未解決の外部記号があることは, このロード・モジュールは, さらに別のモジュールとも結合されなければならないことを暗示している. ともあれ, ロード・モジュールが, 外部記号ディクショナリーをもっていることは, 目的モジュールと同様, ロード・モジュールも連係編集プログラムの入力として使えることを意味している. 別の目的モジュールやロード・モジュールを結合することもできるのである. なお, 連係編集プログラムの詳細については, 後述する.

全ての外部参照記号の解決が行われると, ロード・モジュールは実行可能なプログラムとなる. 制御プログラムが, 主記憶装置を割り当てた上で, ロード・モジュール・ライブラリーから該当プログラムをロードして実行を開始させるのである.

3. モジュールの結合時点と方法

モジュール単位で作成したプログラムは, 翻訳の段階, 連係編集の段階, 実行の段階のそれぞれで結合することが可能である.

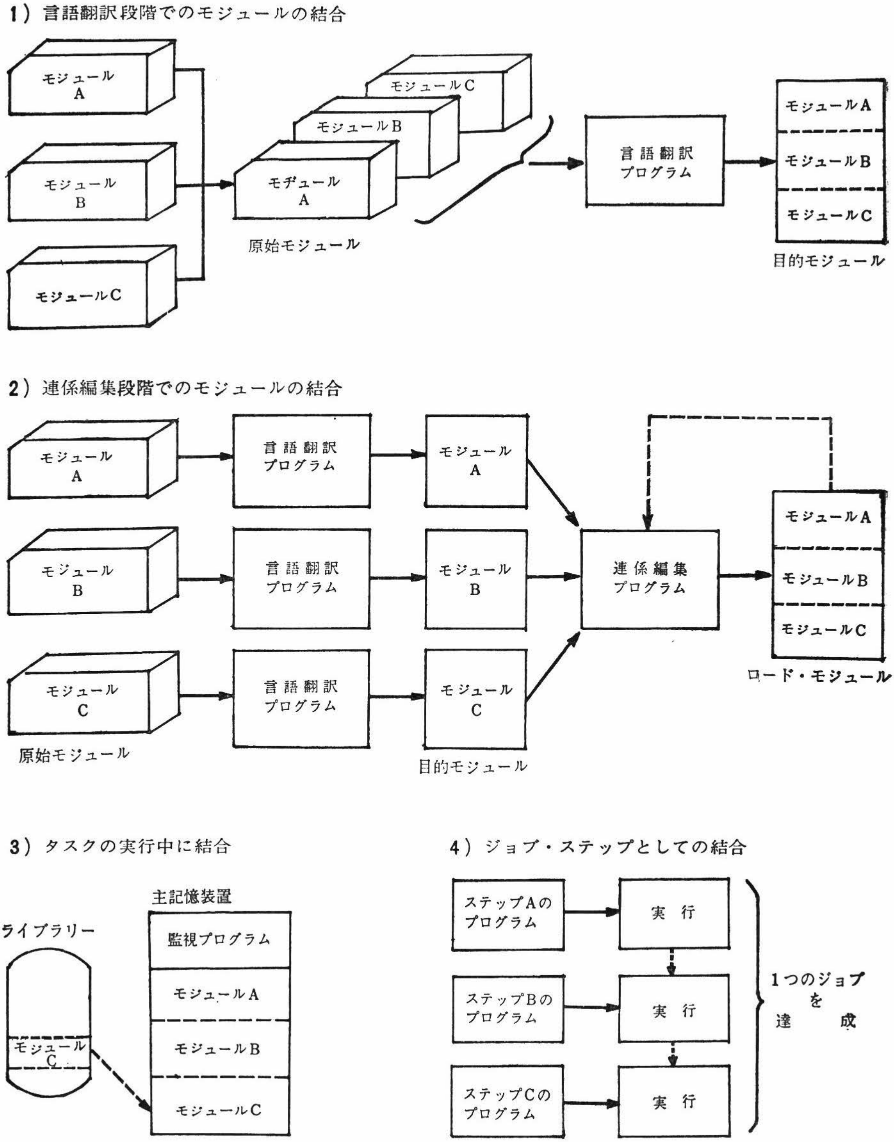
さらに, モジュールの結合とはいえないが, その変形としてジョブの中に複数個のジョブ・ステップを設けることによって, あたかも複数個のプログラムが結合されたと同じ状況を作り出すこともできる.

第3図に示したのが, モジュールの結合の時点とその方法である.

1) 言語翻訳段階でのモジュールの結合

言語翻訳にさきだってプログラムを作成するさい, 1つの仕事を分割してプログラムを作成する. プログラムの作成者は, 1人であっても複数人であってもよい. 複数個のモジュールのうちの1つのモジュールだけを主サブルーチンとみなして作成し, 他はサブルーチンとして作成すればよい. 全てのモジュールは, 原始モジュールとして一括して言語翻訳を行うのであるから, モジュール間での制御権の受け渡しやデータの受け渡しは, 通常のサブルーチンとの間でのやりとりと全く同じ方法でかける.

サブルーチンへのブランチは CALL とか BALR 命令(ブランチ連係命令)



第3図　モジュールの結合時点

を使えばよいし，データの受け渡しには，パラメータによる受け渡しやレジスターによる受け渡しでよい．特別な場合には，直接記号で参照してもよい．

一括して言語翻訳プログラムの入力とすることから，各モジュールは同一言語で作成しなければならず，またある部分の手直しのためには全体を再翻訳しなければならないといった不都合がある．

2）連係編集プログラム段階でのモジュールの結合

言語翻訳プログラムで出力された複数個のモジュールを入力して1つのロード・モジュールに結合してしまう．目的モジュールのもとになった原始モジュールは，いかなるプログラム言語でかかれていてもさしつかえない．なぜなら，原始モジュールが，アセンブラー言語で書かれていようとも，PL/I で書かれていようとも，COBOL や FORTRAN で書かれていても，言語翻訳プログラムで出力された目的モジュールの形式は全く同じであるから．各原始モジュールは，独立したサブプログラムとして書かれる．ただし，各モジュール間で参照される命令やデータは，外部参照記号としての定義が必要である．その他は，原始モジュールの結合でふれたようなサブルーチンの結合方法と同じである．

なお，すでにふれたように連係編集プログラムの入力モジュールとしては，目的モジュールだけでなくロード・モジュールを使用することもできる．

3）タスク実行中のモジュールの結合

連係編集を全く別個に行ったロード・モジュールをプログラム（正式にはタスク）の実行中にダイナミックに呼び込んで実行することもできる．

プログラム全体が一度に主記憶装置に収容しきれない場合の対拠の仕方として，オーバレイの手法がある．ところが，普通のオーバレイでは，どの部分とどの部分をどのようにオーバレイするかあらかじめ計画を立てて主記憶装置の割り振りの仕方を定めておく必要がある．

この方法では，関連するモジュールを一括して連係編集しておく必要がある．ところで，データ通信のプログラムのように各種各様のデータが入力され，

かつ，その入力順序の予測が全く行えないような複雑でかつ例外事項が多発する仕事においては，あらかじめ計画的なオーバレイ構造を設計することは非常にやっかいなことになる．このような仕事においては，その実行中に必要となったモジュールを主記憶装置の空いている場所にダイナミックにローディングして処理できた方がよい．

この種の要求を満すために動的ローディングによるモジュールの結合ができるようになっている．後述するように， LINK 命令， XCTL 命令， LOAD 命令あるいは ATTACH 命令などを使うことによって，実行中に必要なモジュールをロードして処理することができる．

4）ジョブ・ステップとしての結合

あるマスター・ファイルを使って，集計順序が異なる二種類の報告書を作成する仕事があるとする．まず最初に一方の集計順序に合わせてマスター・ファイルのレコードを分類する（ステップ1）．つぎに，報告書を作成するプログラムが実行される（ステップ2）．このあとでマスター・ファイルを別の集計順序に合わせて再分類する（ステップ3）．最後にもう一つの報告書を作成する（ステップ4）．

このようなジョブを想定した場合，この仕事全体を1つのプログラムで実行することも可能であるが，上に示したステップ順にプログラムを分割した方がよいかも知れない．分割するとステップ1と3では，オペレーティング・システムにすでに組み込まれている分類サービス・プログラムがそのまま使える．それにジョブ・ステップとして分割しておいても，見かけ上は1つのプログラムが実行されたのと全く同じ状況を作り出すし，出力される結果にも変りない．ジョブ・ステップに関しては，「ジョブ管理の諸機能とその構造」の章で詳述するが，実質的には，いくつかのサブプログラムを結合して実行するのと同じ効果が得られるのである．

三. プログラム構造と制御プログラムの働き

解決すべき問題が与えられると，現有のコンピューティング・システムで使えるプログラム資源と機械資源，とりわけ，使用可能な主記憶装置の大きさや入出力装置の数と種類の範囲内で最大限の効果と効率を得ることのできるプログラムの設計が必要となってくる．プログラム設計者は，各種制約条件を勘案したうえで一番望ましいプログラム構造とプログラムの分割作成の手法を選択することになる．したがって，あらかじめオペレーティング・システムのもとで許される基本的なプログラム構造とプログラム分割手法に熟知しておくことが要求される．ともあれ，プログラムをいくつかのモジュールに分割して作成したとすると，それぞれのモジュールは言語翻訳プログラムで処理し目的モジュールに変換される．目的モジュールは，連係編集プログラムの入力となり処理されることによって実行可能なロード・モジュールとなる．この連係編集プログラムの段階で，他の目的モジュールやロード・モジュールを結合して，新たなロード・モジュールにすることもできる．

出来上ったロード・モジュールは，ライブラリーに登録される．登録されたロード・モジュールを実行するためには，ジョブ制御ステートメントを使って主記憶装置に読み込ませる．ジョブ制御ステートメントでは，単にロード・モジュールの名前だけでなく，そのロード・モジュールが必要とする入出力データで使用する装置やデータの編成方法なども示す．いわゆるジョブ・ステップとしての記述をジョブ制御ステートメントで行うのである．そのジョブ・ステップが読み取られ，入出力装置や主記憶装置の割り当てが行なわれると，指定されたロード・モジュールがライブラリーから割り当てられた記憶域にロードされる．ロード・モジュール全体が１回でローディングされる場合もあるし，そのロード・モジュールの一部分のみがロードされ，別の部分は必要のつどオーバレイ形式でローディングされる場合もある．さらには，主記憶装置にロードされ実行されているロード・モジュールが全く別のロード・モジュールを呼び出すこともできる．全く別のロード・モジュールとは，目下実行中のロード・モ

ジュールとは別個に言語翻訳を行い，連係編集プログラムで処理されたモジュールであることを意味する．

このプログラム実行の時点にどのようなローディング方式を取らせるかによって，プログラム設計構造が異ってくる．プログラムのローディングと実行時の形態によって，つぎの4つのプログラム構造をとらせることができる．

- 単純構造のプログラム
- 計画的オーバレイ構造のプログラム
- 動的逐次処理構造のプログラム
- 動的並列処理構造のプログラム

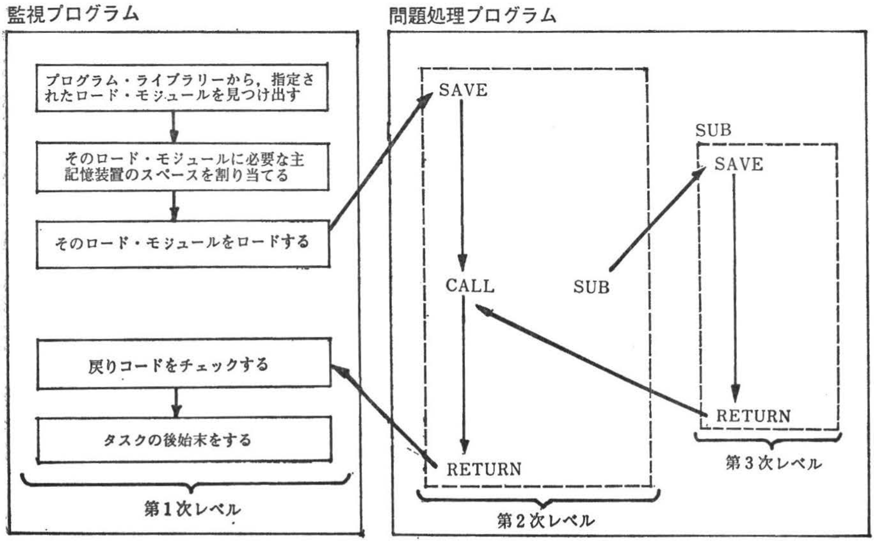
以下，それぞれのプログラム構造と監視プログラムの関係について説明してゆこう．

1.　単純構造プログラムと監視プログラムの働き

単純構造プログラム（simple structure program）は，字義通りわれわれが普通に作成するプログラム構造である．ある1つの仕事を実行するために必要な全てのサブルーチンやサブプログラムが1つのロード・モジュールに完全に包含されている．一度にロード・モジュール全体が主記憶装置にロードされ，そのモジュールだけで仕事の処理が完全に実行される．もちろん，そのモジュールの実行中に割り込み現象によって監視プログラムがしばしば介入することはあるが，別のモジュールやセグメントがローディングされて，制御権がそれらに移るといったことはあり得ない．

ところで，サブプログラム（モジュール）間の結合はどのようにして行うのであろうか．オペレーティング・システムにおいては，制御プログラムであっても，問題処理プログラムであってもすべて標準的な結合方式で制御権の受け渡しを行っている．このことは，モジュラー方式では，極めて重要な要素である．オペレーティング・システムを使用者の各種要求に見合ったプログラム構成で組み立てることができるのは，全てのプログラム・モジュールが標準的結合方式で作られているからである．制御プログラムのあるモジュールを捨て別のモジュールを選択しても，システム全体をそこなうおそれがでないのは，た

とえ１つのモジュールが欠けても，すべてのモジュールが相互に標準的インターフェイスをもっているので支障が生じないのだ．また，われわれが完成したプログラムの一部手直しの必要に迫られたときも，関連するモジュールだけに手を加えて，モジュールを置きかえるだけで，そのプログラムが充分機能するのも実は標準的結合方式をとっているからである．それに，現存のオペレーティング・システムにどんどん新たな機能が組み込めるのも標準的結合方式のおかげである．つまり，柔軟性のあるシステムを指向するさいには，システムを構成する各要素（モジュール）間の標準的連絡が重要なのである．



第４図　プログラムの標準的結合

第４図に示したのがオペレーティング・システムでのプログラム間の標準的結合の基本である．

監視プログラムも沢山のプログラム・モジュールからなり立っているのであるが，ここでは，監視プログラムと問題処理プログラムの関係でプログラムの標準的結合方式をみることにした．監視プログラムは，まずジョブ制御カードで指定されたプログラム（実は，ロード・モジュール）をロード・モジュール

・ライブラリーから捜しだす．ライブラリーのインデックスを検索すると該当ロード・モジュール名に一致するインデックスをつかむことができる．そこにはそのロード・モジュールが直接アクセス装置の何番地から何バイトの大きさで存在しているといった情報が示されている．そのロード・モジュールの大きさがわかれば，主記憶装置の空いているスペースを何バイト確保すべきかが決まる．監視プログラムはスペースを見つけだしたうえで，インデックスに示されている番地をもとに該当ロード・モジュールを選び出してローディングする．主記憶装置の現在未使用の場所を見つけて，ローディングするわけであるから，始めから主記憶装置の何番地から，このモジュールがローディングされるかは予想出来ないことである．したがって，ロード・モジュールは**再配置ローディング**（relocatable loading）されざるを得ない．つまり，ロード・モジュールは再配置可能プログラムである．ともあれ，該当モジュールがロードされると，監視プログラムの制御権は，問題処理プログラムに移る．その制御権が移る先には，SAVEマクロ命令がなければならない．この命令によって，上位レベル（この場合は監視プログラム）のプログラムが使用していたレジスターなどの内容を指定された記憶域に保存しておくのである．再び上位プログラムに制御権が舞いもどったとき，元の状態に復元できるようにしておく．われわれが，COBOL や PL/I のような高級言語でプログラムを作成する場合には，SAVE マクロや後述の RETURN マクロを使用することはないが，これら言語で書いたプログラムを言語翻訳するとこの種の機能を果す機械語命令が出来上って，標準的結合が計られていることを銘記されたい．

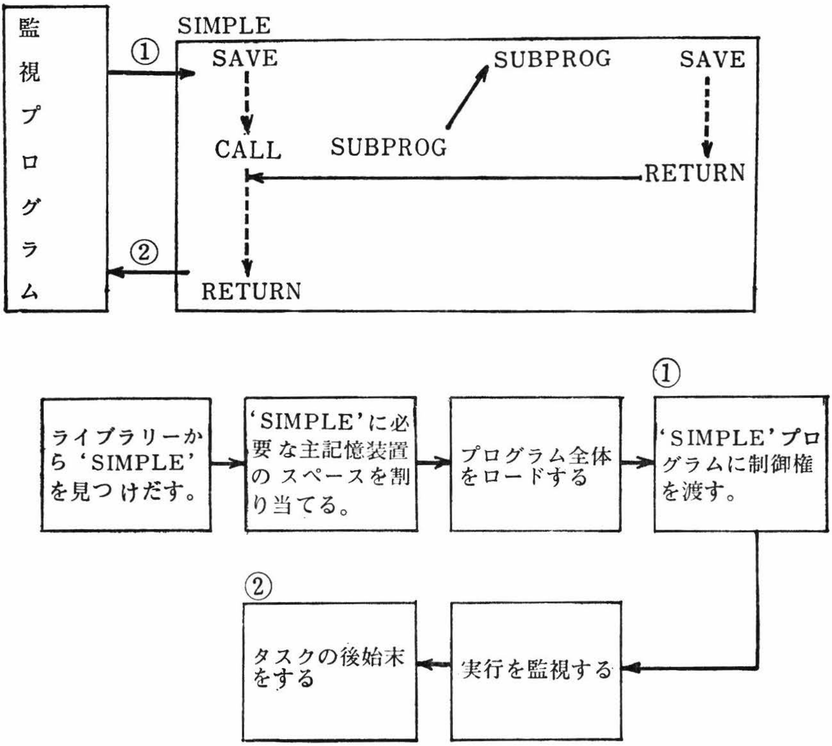
このプログラムの中で CALL 命令を使ってサブプログラムを呼ぶと，1つ下のレベルとみなされるサブプログラムに制御権が渡る．サブプログラムの入口点の命令も SAVE 命令でなければならない．第2次レベルのプログラムがブランチする時点に使用していたレジスターなどの内容をそのまま保存しておくためである．このサブプログラムでの処理が終ってもとのプログラムに戻るときには，RETURN マクロ命令を使う．この命令を与えると，保存しておいたレジスターなどの内容をもとにもどして，つまり上位レベルのプログラムがブランチする直前の状態にして，そのプログラムに制御権をかえす．この

RETURN 命令で**戻りコード**（return code）をつけてかえすこともできる．戻りコードというのは，そのサブプログラムの実行がどういう条件で終了したのか，何らかのエラーがあって終了したのか，それとも，無事終了したのか，といった条件コードを上位レベルのプログラムにかえす目的に使われる．上位レベルのプログラムでは，戻りコードを分析して所要のルーチンをとらせることができるのである．

監視プログラムに制御権をもどすのにも RETURN 命令を使う．機能は，前述の場合と同様である．監視プログラムに制御権がかえされると，戻りコードを分析したうえで，問題処理プログラムで実行したタスクの後始末，つまり，使用した入出力データ・セットの取りはずしの指示だとか，メッセージの印刷だとかを行う．

このように監視プログラムと問題処理プログラム，あるいは，問題処理プログラムの中での上位レベルのプログラムとサブプログラム間では，標準的な制御権の授受が行なわれるように設計されているのである．第4図とこれまでの説明でお気づきのように，監視プログラムは第1次レベルのプログラムともいうべきものである．監視プログラムの取扱いからすれば，問題処理プログラムはサブプログラムと同じになる．ちょうど第2次レベルのプログラムが，第3次レベルのプログラムをサブプログラムとみなすと同様の処理をしているのである．したがって，仮りに，第2次レベルのプログラムがなくて，第3次レベルのプログラムだけがローディングされたとしても，監視プログラムからの制御権の受け渡しは全くスムーズにゆくのである．SAVE，RETURN という標準的結合方式をとっているからである．このことから，サブプログラム間の組みかえが容易に行いうる理由を了解していただけることと思う．さらに重要なことは，このような標準的な制御権の授受を利用することによって，ある問題処理プログラムを別の問題処理プログラムで呼び込んで利用することもできるということにつながる．

たとえば，COBOL や PL/I で書いたプログラムの実行中に汎用性のある分類プログラムを呼び込んで活用するといったことが可能になるのは，このような標準的インターフェイスが保たれているからである．なお，このことにつ



第5図　単純構造のプログラムと監視プログラムの働き

いては，動的構造のプログラムで詳述する．

プログラムの標準結合方式に紙面をとられたが，主題にもどることにしよう．単純構造のプログラムは，ただ1つのロード・モジュールからできておりそのモジュールは，一度に主記憶装置に読み込まれる．そして，そのモジュールだけで仕事の処理を遂了するような構造でできているのが単純構造のプログラムであった．しかし，先述の標準結合方式の考えからすれば，単純構造のプログラム自身が他のロード・モジュールによって動的に呼び出されて利用されることもある．

第5図に単純構造のプログラムと監視プログラムの関係を示した．SIMPLE と名付けたロード・モジュールは，監視プログラムによって1度にローディン

グされたものと仮定している．1つのロード・モジュールが複数個のサブプログラムから成り立っていても一向に不思議はない．なぜなら，連係編集プログラムによって複数個の目的モジュール（それぞれの目的モジュールをサブプログラムと考えてよいのであろうが）を結合して1つのロード・モジュールを作ることができるからである．さらには，すでに連係編集プログラムで処理ずみのロード・モジュールを別のロード・モジュールや目的モジュールと結びつけることだってできるからだ．このような場合には，確かにプログラム自体は複雑である．しかし，ここで説明しているプログラム構造は，ロード・モジュールのローディングと実行形態を基準に分類しているので，最終的に1つのロード・モジュールにまとめられ，かつ1回のローディングで仕事が完全に処理できるようなプログラム構造なら，どんなに複雑なプログラムでも依然として単純構造と考えざるを得ない．

第5図の場合，1個のサブプログラムを CALL しているが，このプログラム全体が1つのロード・モジュールとしてローディングされ，これだけのプログラムだけで仕事を処理するものと仮定している．当然監視プログラムは，'SSIMPL' というロード・モジュールをプログラム・ライブラリーから捜し出し，主記憶装置のスペースを割り当てて，ローディングしたうえで，制御権を問題処理プログラムに渡す．あとは，問題処理プログラムの中でデータ処理を続行してゆく．サブプログラムと制御権の授受を行う場合でも特に監視プログラムの援助を必要としない．ただし，監視プログラムは，入出力要請だとか，その他の割込みサービスを行うためにたえずプログラムの実行を監視しているのである．処理が終れば，RETURN 命令によって監視プログラムに制御権が移り，そのプログラムが実行したタスクの後始末をする．

2. 計画的オーバレイ構造のプログラムと監視プログラムの働き

計画的オーバレイ構造 (planned overlay structure) のプログラムも単純構造のプログラムと同じように1つのロード・モジュールである．しかし，単純構造のプログラムとちがって，1度にプログラム全体が主記憶装置に読み込まれない．最初にローディングされるのは，**ルート・セグメント**と呼ばれる部分

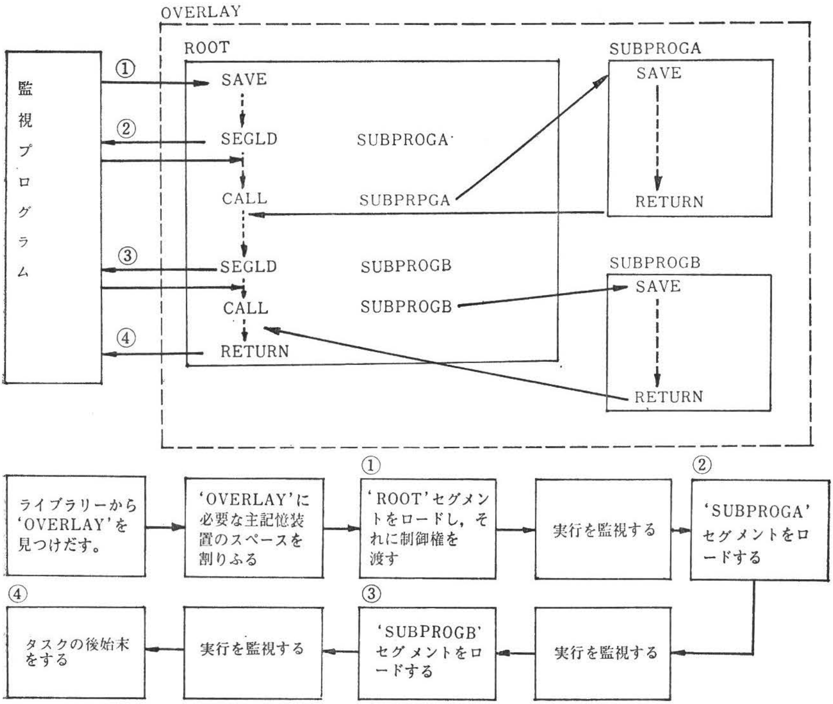
だけである．**セグメント**（segment）とは，主記憶装置にローディングするさいの最少の論理的単位である．したがって，セグメントがさらに細分化されて，何回かに分かれてローディングされることはない．ともあれ，計画的オーバレイ構造のプログラムは，必ず複数個のセグメントから構成されており，そのうちのルート・セグメントだけがジョブ制御ステートメントの指定にしたがって最初にローディングされる．そして，このセグメントだけは，プログラムの実行中常時主記憶装置に留まっている．他のセグメントは，プログラム実行過程で必要に応じて，オーバレイ形式でローディングされる．

このオーバレイ構造は，プログラム設計のさいにあらかじめきめておき，連係編集プログラムに対してどのような構造でオーバレイをするかを OVERLAY制御ステートメントで指示しておく．この指示にしたがって，連係編集プログラムが，プログラムをセグメント単位のオーバレイ構造に形作ってくれる．このように，あらかじめオーバレイ構造を設計して連係編集プログラムに指示を与えておくので，計画的オーバレイ構造と呼ばれるのである．これは，後述する動的オーバレイ形式に対応する言葉として使われている．

オーバレイ構造の指定の仕方については，「連係編集プログラム」の項でふれることにして，ここでは，簡単な例をもとに計画的オーバレイ構造と監視プログラムの関係について説明しよう．

第6図がその例である．破線で囲まれた3つのセグメントが1つのロード・モジュールを構成するものとする．破線のうちで，'ROOT' と名付けられているのがルート・セグメントであり，最初にこの部分だけがローディングされている．他のセグメント 'SUBPROGA' と 'SUBPROGB' は，プログラムの実行過程で必要に応じてローディングされる部分である．'SUBPROGA' と 'SUBPROGB' は同時点で必要とはしないので，同一主記憶域にオーバレイ形式で呼びこまれる．

監視プログラムがライブラリーから計画的オーバレイ構造のプログラムを見つけだすと，プログラム全体に必要となる主記憶装置の大きさは連係編集プログラムの処理結果の情報としてそのロード・モジュールのインデックスに貯えられているので，所要の主記憶装置のスペースを割り当てることができる．監



第6図　計画的オーバレイ構造と監視プログラムの働き

視プログラムは，まずルート・セグメントの部分だけをローディングし，それに制御権を委譲する．プログラムの結合方式は，計画的オーバレイ構造であっても，単純構造の場合と同様標準的結合方式にしたがう．ルート・セグメントのプログラムを実行している過程で SEGLD 命令にぶつかると，制御権は監視プログラムに渡る． SEGLD 命令で，どのセグメントをローディングする必要があるかを指示しているので，監視プログラムは，そのセグメントをライブラリーからローディングすることを指図したあとで，“ルート・セグメント”の SEGLD 命令のつぎの命令から実行を再開する．そのセグメントがローディングされているのに併行して，つぎの命令から実行されてゆくのである．このようにあらかじめ必要となるセグメントが判明する場合には，SEGLD命令でローディングを少し早めに行わせる．ところが，ロードすべきセグメ

ントが早めに判明しないときには，ローディング作業とプログラムの実行という仕事を併行操作することは期待できない．このような場合には，SEGLD といった特別な命令を使う必要は毛頭ない．CALL 命令だけで充分である．SEGLD 命令がないと，CALL 命令にぶつかったら，ただちに監視プログラムに制御権が渡り，そのセグメントのローディングが完了した時点で，制御権がロードされたセグメントに渡される．そのセグメントからルート・セグメントへの制御権の返還は，RETURN 命令によって標準的に行われる．

別のセグメントの要請（SEGLD 命令あるいは CALL 命令）が出ると，再び制御権は監視プログラムに移り，不用になった ‘SUBPROGA’ セグメントが占めている記憶域にオーバレイの方法で，‘SUBPROGB’ のセグメントがローディングされる．

このようにあらかじめ予定された形式にしたがってオーバレイが行なわれていくようなプログラム構造が計画的オーバレイ構造である．計画的オーバレイ構造のプログラムの実行にさいして，オーバレイ構造がどのようになっており各セグメントを主記憶装置のどの部分にローディングし，そのあとでどの命令に制御権を移譲すべきかなどの監督を行っている監視プログラム部分を特に**オーバレイ監視プログラム**（overlay supervisor）と呼んでいる．

3. 動的逐次処理構造のプログラムと監視プログラムの働き

動的構造（dynamic structure）では，プログラムの実行中に2つ以上のロード・モジュールが呼び出される．それぞれのモジュールは，全く別個に連係編集されたものである．したがって，モジュール間の連絡の仲介役として監視プログラムが援助的役割を果すことによって，両者間では標準的な結合方式がとられることになる．

ある種の適用業務では，単純構造や計画的オーバレイ構造のプログラムが適当でない場合がある．1つのプログラムが沢山のサブプログラムから成り立っており，かつそれらの実行の順番が処理すべきデータ・レコードの内容を調べないと決定出来ないようなプログラムである．このようなプログラムにおいては全てのサブプログラムを1度に主記憶装置に呼び込んでおくことは実際的で

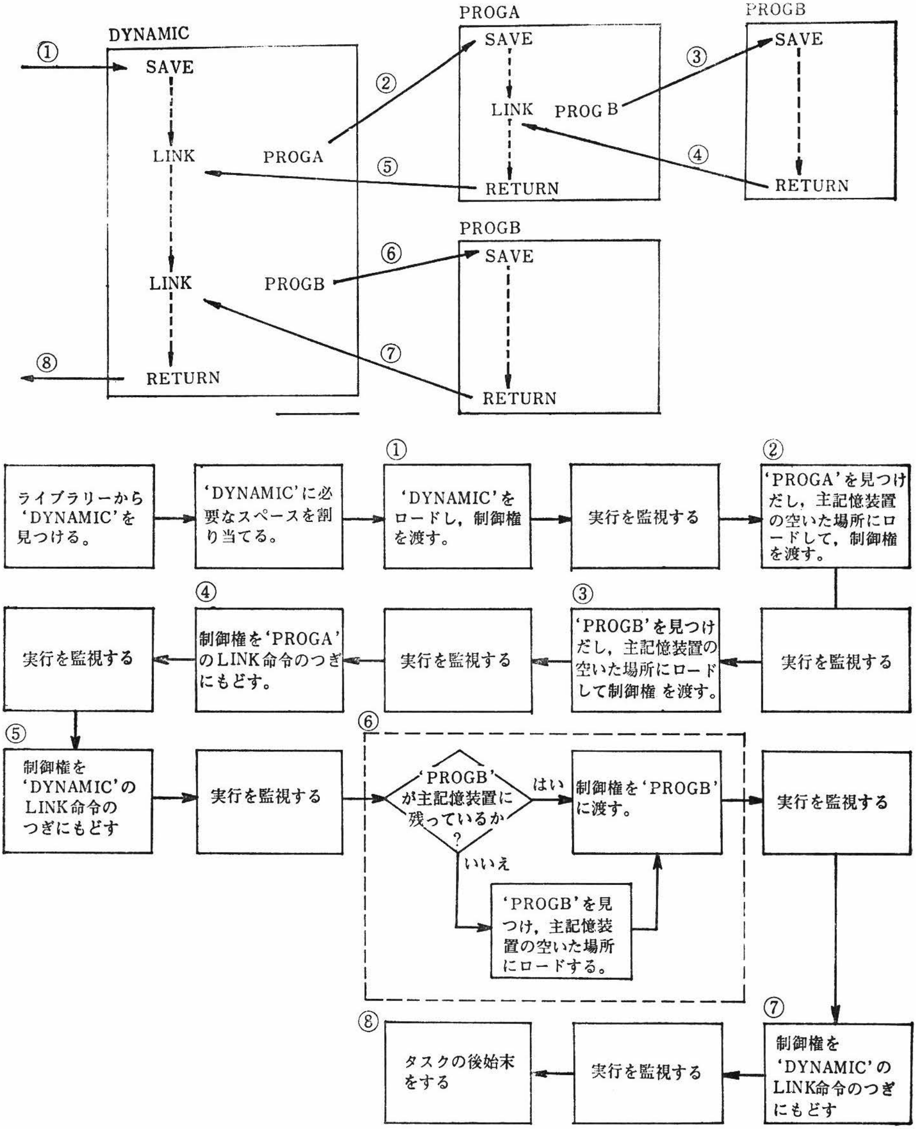
はないし，一方，サブプログラムの使用順序がランダムであるため計画的オーバレイ構造をとることは不経済でかつ不可能に近くなる．このような性格のプログラムに対して，オペレーティング・システムが提供している機能が，サブプログラムをダイナミックに呼び込む方法，つまり，あるロード・モジュールが実行されている最中に別のロード・モジュールを呼び込む方法である．

オペレーティング・システムで，このような動的呼び出しが可能になるのはつぎのような機能を制御プログラムがもっているからである．まず，ロード・モジュールの名前がわかれば，その名前でモジュールを検索し呼びだすことが制御プログラムの普通の機能になっているからであり，かつ，それに必要な主記憶装置の要求があればそのつど動的に割当てることができるからである．

これらのことは，制御プログラムがコントロールを行う際の最も重要な単位がタスクであり，そのタスクを正常に実行してゆくための資源（主記憶装置や種々のプログラムおよび CPU や各種入出力装置など）は，必要に応じて何時でも要求することができるという基本的思想に合致している．

そのうえ，どのモジュールでも１つのサブプログラムとして標準的結合方式で実行できるようになっている．したがって，監視プログラムは，この標準的な規則にのっとってサブプログラムへの移行とサブプログラムからの復帰を行なう仲介的役割を果すことができる．実行中のロード・モジュールから別のロード・モジュールを主記憶装置に呼び込むためのマクロ命令として， LINK, XCTL および LOAD の３つのマクロ命令がある．

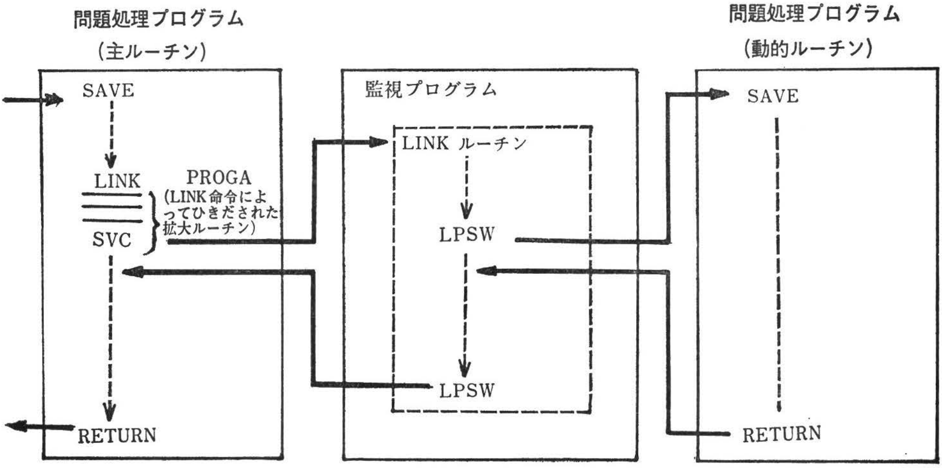
LINK の働きと監視プログラムの機能の仕方を示したのが**第７図**である．３つのロード・モジュール ‘DYNAMIC’, ‘PROGA’, ‘PROGB’ のうち ‘DYNAMIC’ だけが最初にロードされたモジュールである．このモジュールがロードされたあと，制御権は，監視プログラムから ‘DYNAMIC’ に移る．LINK マクロ命令にぶつかると制御権は再び監視プログラムにもどされる．監視プログラムは， LINK 命令のオペランドで指定されたロード・モジュールをライブラリーから見つけだし，主記憶装置のスペースをそのモジュールに割り当てたうえでローディングし，制御権をそのモジュールに渡す．モジュールの実行が終ると制御権は RETURN 命令によって一たん監視プログラムに戻され



第7図　動的逐次処理構造と監視プログラムの働き

る．監視プログラムは，そのモジュールが占めていた記憶域を他の用途のための空きの場所として一応解放する．ただし，他の用途でその場所が占有されるまで，使われたモジュールはそのまま記憶されている．つまり，監視プログラムはこの場所が空いているということを登録したにすぎないのであって，記憶されているモジュールを破壊したわけではない．したがって，第7図において，'PROGB' が2回目に呼ばれたとき，もとのままの状態でその場に残っていれば，このモジュールは，あらためてライブラリーから取り出されることはない．ただし，前のものが再使用できるための条件としては，後で述べるようにこのモジュールに"再使用可能"というプログラム上の性質をもたせておくことが必要である．

ところで，LINK マクロ命令が機械語命令におきかわると，その中には SVC 命令がふくまれている．



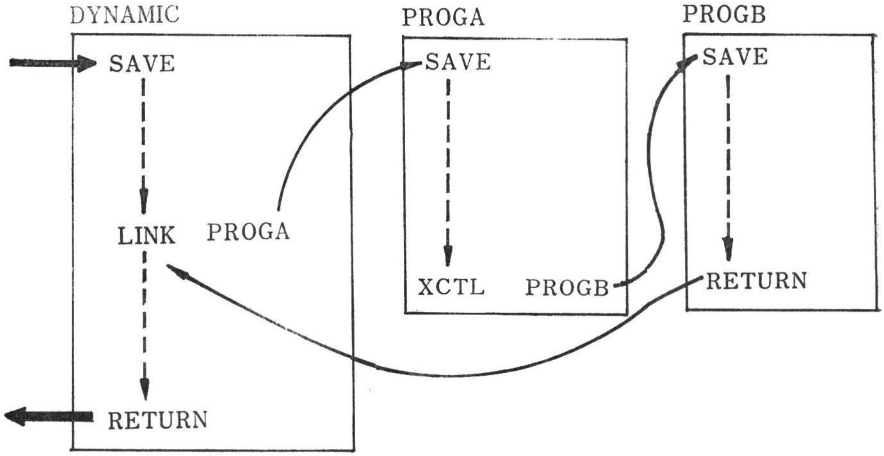
第8図　LINK 命令と監視プログラムの援助機能

第8図にその様子を示した． SVC 命令は監視プログラム呼び出しの割込みを引きおこす命令である．それゆえ，制御権は，この割込みを通して監視プログラムに渡る．監視プログラムの中の LINK ルーチンでは，指名されたロード・モジュールをライブラリーから捜しだしたうえで，主記憶装置のスペースを割り当てたあとそのモジュールをローディングするわけだ． そののち， LP

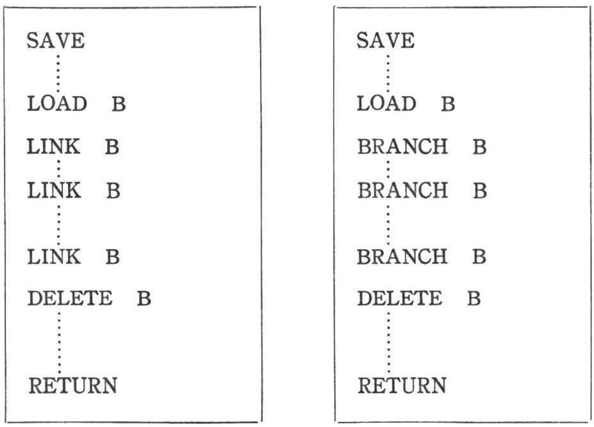
SW 命令によって，制御権を動的にローディングしたモジュールにひき渡す．そのロード・モジュールが実行されたあと RETURN 命令によって再び監視プログラムにもどされ，そのモジュールが占めていたスペースを解放したあとで LPSW 命令を出す．この命令によって，今度は，LINK 命令を出したモジュールに帰るのである．このように動的構造においては，モジュール間の制御権の移動には監視プログラムの援助機能が必ず必要となるのである．このことは，XCTL 命令や LOAD 命令においても同じである．なお，これまでみてきたように LINK 命令によるモジュールの結合においては，一方のモジュールに入出力などの待ちが生じたとしても，他方のモジュールがその待ち時間を利用して実行されるということはない．つまり，モジュール間で並列的に実行が進むことはなく，いつでも逐次的実行である．したがって，LINK，XCTL あるいは LOAD 命令を使うプログラム構造は，**動的逐次処理構造**（dynamic serial structure）と呼ばれるのである．この点が後述する動的並列処理構造と異っている．

第9図に示したように RETURN 命令の代りに XCTL マクロ命令を使うと別のロード・モジュールを呼びこみ，制御権はそのモジュールに移ってゆく．標準的な結合方式の規則が守られ，監視プログラムが行なう機能は LINK 命令の場合と同様である．ただし，XCTL 命令の場合には，その命令によって制御権が監視プログラムに渡ると，XCTL 命令をだしたモジュールが占有していた主記憶域はただちに解放され，再びそのモジュールに制御権がもどされることはない．しかも，モジュールの制御レベルは，呼び出したモジュールと呼び出されたモジュールともに同一レベルとみなされるので，呼び出されたモジュールで RETURN 命令を出すと，制御権は監視プログラムを経由したうえで，LINK 命令のつぎに帰るのである．

LOAD マクロ命令の主要な目的は，あるロード・モジュールが頻繁に使用される場合に効率よく対処するという点にある．LOAD 命令を使って，モジュールをロードすると，DELETE 命令によってその記憶域が解放されるまでそのモジュールがそのまま保持される．LINK や XCTL によって呼び出された場合には，そのモジュールが実行されると直ちにそれが占める記憶域が解放



第9図　XCTL 命令の機能



(LINK 命令との組合せ)　(ブランチ命令との組合わせ)

第10図　LOAD 命令の機能

されたのであるが，LOAD 命令を使うと DELETE されるまでそのモジュールが生きているので，再三再四ローディングするという時間的手間をはぶいて繰返し利用できる． LOAD 命令には，モジュールに制御権を引き渡す機能は

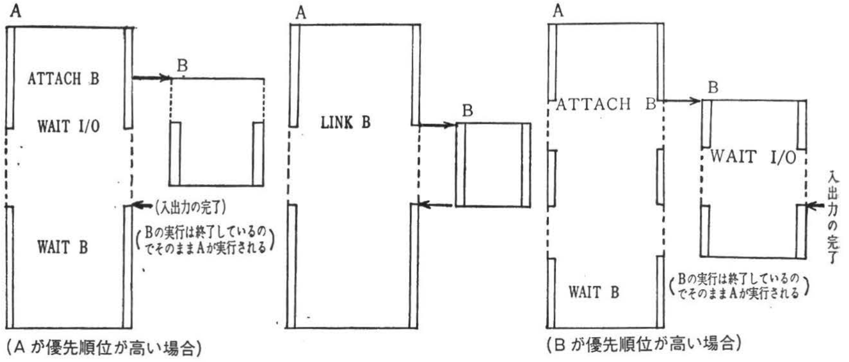
ないので，ローディングしたあと LINK または普通のブランチ命令を使って，そのモジュールに制御権を渡す方法をとる．LINK 命令が使えるのは，LINK したモジュールが主記憶装置に残されているかどうかチェックしたあとで，そのモジュールが主記憶装置に存在しないときだけローディングしてくれる機能があるからだ．あらかじめ，LOAD 命令でロードされておれば，LINK 命令で再びロードすることなく制御権が移ることになる．なお，LOAD 命令で引き出されるモジュールは，“再使用可能”という性質がもたされたプログラムであることが必要である．

4.　動的並列処理構造のプログラムと監視プログラムの働き

これまで説明してきた3種類のプログラム構造では，プログラムの実行の順序は，すべて逐次的であった．ところが，これから述べる ATTACH マクロ命令を用いれば，その命令が実行されると新しいタスク（一般に**サブタスク**と呼ばれるが）が作り出され，ATTACH 命令をもったロード・モジュールと並列的に実行されてゆく．これ以外の点では，ATTACH 命令は LINK 命令とよく似ている．LINK 命令があくまで逐次的実行を要求するのに対して，ATTACH 命令は並列的実行を要求する．しかも，ATTACH 命令は新しいタスクを作りだすのに用いられるので，それだけ多くの監視プログラムの時間を必要とするし，制御の仕方も複雑になってしまう．したがって，この ATTACH 命令を使って新しいタスクを作り並列実行を行うことができるのは，高度なオペレーティング・システムだけである．

ATTACH と LINK の機能上のちがいを示したのが**第11図**である．真中に示したのが LINK を使った場合である．LINK 命令によって，モジュールがローディングされ制御権が移るとそのモジュールの実行が完全に終るまで，制御権がもとのモジュールにもどることはない．つまり，‘B’ の実行が終了するまで ‘A’ の実行再開は期待できないのである．

一方，ATTACH を使用した場合には，タスクの実行優先権をもったモジュールの実行過程で入出力完了などの待ちが生じたときに，別のモジュールに制御権が移される．優先権の高いタスクのモジュールの待ちが完了すると再び制



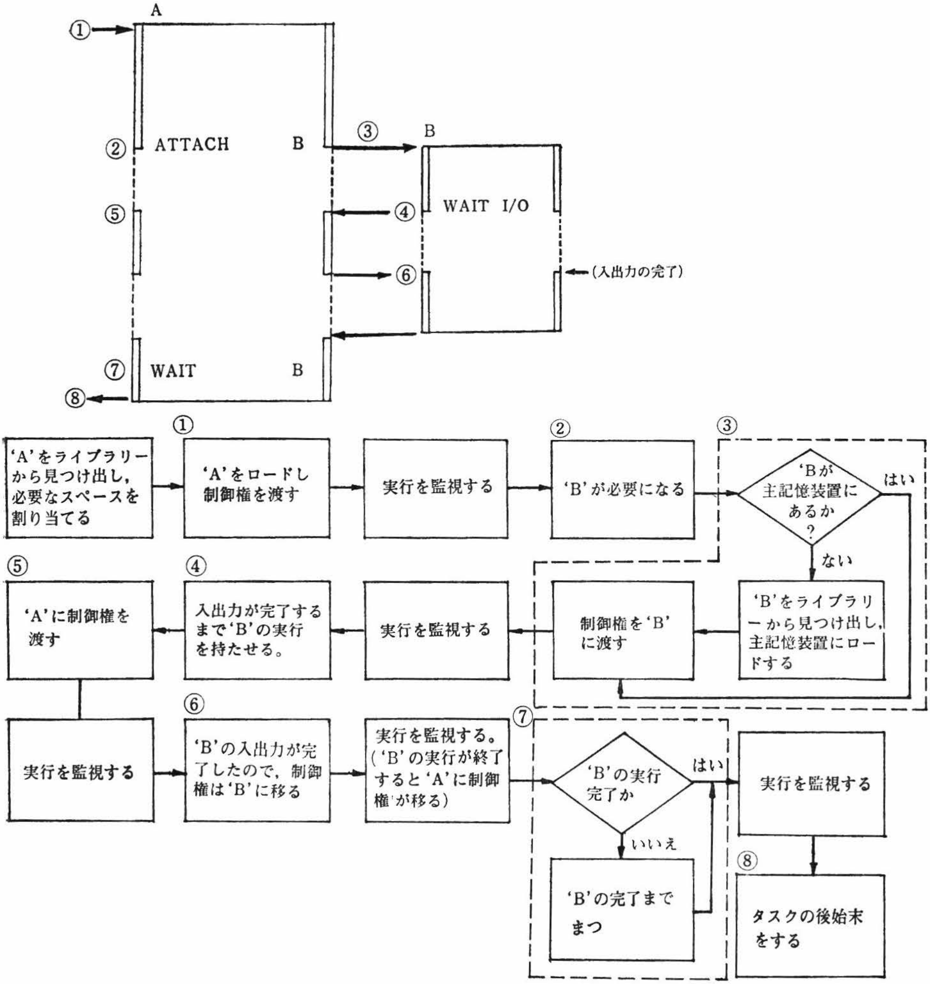
第11図　ATTACH 命令と LINK 命令の違い

御権はもとのモジュールに移譲される．なお，タスクの優先権は，ジョブ制御ステートメントで与えることもできるし，ATTACH 命令で与えることもできる．ジョブ制御カードで与える優先権は，親タスクのそれであり，ATTACH で与えるのはサブタスクの優先権である．**第11図**の左側で示したのは親タスクの方が優先権が高い場合である．したがって，ATTACH 命令で，サブタスクを生成しモジュール 'B' をローディングしても直ちに制御権は 'B' に移らない．そのまま 'A' を実行してゆき，入出力の待ちが生じたときにはじめて 'B' に制御権が与えられて実行が開始される．この例では，'A' の入出力待ちが完了するまえに 'B' の実行は終了してしまっている．

第11図の右側では，サブタスクの方が優先権が高いケースをとり上げている．この場合には，ATTACH 命令でサブタスクが生成され，モジュール 'B' がローディングされると直ちに制御権は 'B' に移る，'B' の実行過程において待ちが生じたときにだけ，制御権は 'A' にかえる．しかし，'B' の待ち状態が解消すると再び制御権は 'B' に移るのである．

このように並列処理が行われているさいのモジュール間の制御権の移動の仲介役を行うのは勿論監視プログラムである．この様子を示したのが**第12図**である．

以上みてきた ATTACH の機能から推察できるように，2つのタスクの間



第12図　動的並列処理構造と監視プログラムの働き

でかなりの併行処理が期待できるとき，つまり，優先権の高いタスクにおいて確実に待ち状態が予想できるときにだけ，ATTACH を使うべきである．この種のことが期待できないとき ATTACH 命令を使用すると，サブタスクの生成や制御権の受け渡しに余分の時間をくわれてしまうので得策でないことになる．したがって，このような場合には，LINK，XCTL あるいは LOAD 命令

を使った動的逐次処理構造のプログラムにすべきである.

5. プログラム作成上の考慮点

これまで4種類のプログラム構造と監視プログラムの関係をみてきたのであるが, ここでプログラム作成上の考慮点についてまとめておこう.

すでにふれたように LINK とか LOAD 命令でローディングされたモジュールがそのまま主記憶装置に残っていれば再びそのモジュールの使用が要求されたとき, そのモジュールが使用できることを指摘した. ただし, その場合には, **逐次再使用可能** (serially reusable) という性質をプログラムに与えておく必要があった. さらに, ATTACH 命令によってよび出されるロード・モジュールを2つ以上のタスクが同時並列的に使用するためには, **再入可能** (re-enterable) という性質をプログラムに与えておく必要がある. 再入可能でも逐次再使用可能でもない性質がもたされたプログラムは**再使用不可能** (non-re-usable) である.

ライブラリーに入っているすべてのロード・モジュールには, 上記3種類のいずれかの性質が与えられている. この性質は, プログラマーが連係編集のさいに指定すべきものである. プログラマーは, あからじめプログラム作成過程で再使用不可能, 逐次再使用可能あるいは再入可能のいずれかの性質に見合ったコーディングを行っておく必要がある.

再使用不可能の性質をもったモジュールというのは, そのモジュールの実行過程でモジュールの一部分を変更してしまうようなプログラムであり, 再びそのモジュールを使用するためには, ライブラリーから再度読み込まないと正しく実行できないものである. ここでいうモジュールの一部とは, 命令だけでなく, 処理に使われるデータ域をも含んでいる.

一方, 逐次再使用可能という性質のモジュールは, そのモジュールの中で自分自身を最初の状態に複元するようにコーディングされたプログラムである. つまり, モジュール中で**初期設定** (initializing) をするように設計しているので, そのモジュールが実行されることによって変更された部分は, 再使用されるまえに最初の状態にもどされる. したがって, 一たんローディングされたモ

ジュールがそのまま残っておれば，タスクの実行の間に繰り返して使用することができるのである．

再入可能のモジュールでは，同一モジュールが同時点に2つ以上のタスクで利用されるので，モジュールが実行されている間どの部分も一切変更や修正が行われないように設計しておく必要がある．つまり，このモジュールは“リード・オンリー”であることが重要である．

プログラム作成において，再使用不可能，再使用可能あるいは再入可能のどの性質をもたせるかによって，それぞれの性質に見合った作成技法を適用することが肝要である．

つぎに，どのプログラム構造を採用すべきかを検討するさいの考慮点についてふれておこう．

まず，可能なかぎり単純構造を選択するようにすべきである．単にプログラミングが簡単であるという理由ではなく，最も効率のよいプログラム構造が単純構造だからである．というのは，単純構造におけるサブプログラムの連係は全く監視プログラムの援助なしに直接制御権の受け渡しが行なわれるので，それだけ実行効率が上るのである．

単純構造の適用が困難な場合には，計画的オーバレイ構造か動的構造のどちらかを選択することになる．計画的オーバレイ構造やロード・モジュールの動的呼び込みのための監視プログラムの機能は，あくまで，その仕事に許容される記憶域を越えるような大きなプログラムの実行の必要性を満すために用意されたものである．したがって，記憶容量の範囲におさまるプログラムをあえて動的構造や計画的オーバレイ構造にする必要はないのである．しかし，両者はそれなりの特性をもっている．計画的オーバレイ構造は，あらかじめ連係編集プログラムによって，セグメント間の相互の参照を直接行うことができるようにしているので，プログラムの実行効率の点では動的構造よりまさっている．さらに，計画的オーバレイ構造では，1つのロード・モジュールをセグメントに分割してローディングするに過ぎないので，ライブラリーの中のモジュールの存在場所は1回でつきとめることができる．ところが，動的構造では，沢山のロード・モジュールから成り立っいるので，モジュールのローディングが要

求されるごとにライブラリーを検索するという手間が生じる．

しかし，計画的オーバレイ構造のこの種の特性も，プログラムが複雑化してくると帳消されてしまう．たとえば，サブプログラムの実行順序の選定が，乱順序に入力されてくるデータの形式によってはじめて可能になるような場合には，あらかじめ論理的順序でオーバレイ構造を定めることはむずかしい．このような条件のもとでは，計画的オーバレイ構造よりも，実行過程で必要に応じて動的に読み込んだサブプログラム（つまり，モジュール）を使用する方法の方がすぐれた解決法となる．そのうえ，ダイナミックにローディングしたモジュールは，異ったタスクから再使用が許されるが，計画的オーバレイ構造のセグメントは再使用できない．

もちろん，同一の問題を解決するのに，動的構造と計画的オーバレイ構造を組合わせてプログラミングすることは一向にさしつかえない．動的に呼び出されたモジュール自身がオーバレイ構造であってもよいし，逆に，計画的オーバレイ構造の中に LINK 命令が使われていてもかまわない．

動的構造を適用するということが決定されたとなると，あとは，**逐次処理構造**か**並列処理構造**（dynamic parallel structure）かのどちらかを選択する問題につきあたる．すでにふれたように並列的構造を選択した方がよいのは，2つ以上のタスクでかなりの併行処理が期待できるときだけである．

以上のことを中心にどのプログラム構造を適用すべきかを検討すべきである．要は，ハードウェア・システムの一定の制約条件のもとで，最大限の処理効率を上げ，かつ，プログラム作成の労力を最小限にするようなプログラム構造を選ぶように努めることである．

四. 連係編集プログラムの機能構造

連係編集プログラム（linkage editor）も，オペレーティング・システムの構成要素の1つである．言語翻訳プログラムやユーティリティー・プログラムと同様に処理プログラムとして位置づけられる．言語翻訳プログラムが，原始プログラム・デックを入力し，それを処理して機械語命令からなる目的モジュールを出力することは，今更いうまでもないことである．これに対して，連係編集プログラムは，言語翻訳プログラムが出力した目的モジュールを入力し，それに処理を加えて，実行可能なプログラム・モジュール，つまり，ロード・モジュールを作り出す機能をもっている．入力モジュールは，単に目的モジュールにかぎらず，すでにロード・モジュールとなっているものでもよい．いくつかのモジュールを結合して，1つのロード・モジュールにまとめ上げてしまうのが，連係編集プログラムである．

そもそも，コンピューターの初期の段階には，連係編集プログラムなんて存在しなかった．アセンブラーにしろ FORTRAN にしろ，これら言語で書き上げたプログラムは，言語翻訳プログラムで機械語プログラムに変換するだけで実行することができた．プログラムがそれほど複雑でなく，プログラムを分割して作成したり，標準的サブルーチンを有効に使うことによってプログラマーの作業負担を少なくするといった強い要請がない時代には，これで十分通用したのである．したがって，オペレーティング・システムが主流を占めない時代，すなわち，第2世代のコンピューターでは，連係編集プログラムの必要性はほとんどなかった．

ところが，プログラムの作成効率の向上，共通的ルーチンの効果的反復利用，プログラム・デバッキングと手直しの効率化あるいは最適プログラム言語を使ってのプログラム・コーディングなどを目的として，プログラムの分割とモジュール化が企図されるようになってくると，それを押し進めるに便利な機能を提供する何んらかのサービス・プログラムの介在が要請されるようになる．ちょうど，第1世代のコンピューターにおいて機械語でプログラムのコーディ

ングを行っていたとき，もっと簡単なプログラムの作成ということから言語翻訳プログラムが誕生したのと同様である．

連係編集プログラムの機能があるからこそ，われわれプログラマーは，１つのプログラムをいくつかのサブプログラム部分に分割して作成することができる．それぞれのプログラム部分は，その処理内容に一番ふさわしいプログラム言語を利用してコーディングできる．各サブプログラムは，それに見合った言語翻訳プログラムを使って別々にアセンブルないしはコンパイルする．これらの目的モジュールを結合して，１つの実行可能なロード・モジュールに仕立て上げるのが，連係編集プログラムの主要目的といえる．別々に作成した目的モジュール相互間でのデータや命令のやり取りをうまく解決し，あたかも一緒に言語翻訳を行ったかのごとき完全な１つのロード・モジュールとすることが，連係編集プログラムに要求される主要な働きである．

「連係編集プログラム」という言葉は，今では普通に使われるようになっている．しかし，初期の頃は議論百出したものだ．もともとの言葉が「Linkage Editor」という用語なので，「連結プログラム」がよいとか「結合編集プログラム」だとか「リンケージ・エディター」のままがよいとか，いろいろの案が出た．「連係編集プログラム」という用語は，舌をかみそうな表現なので馴染めないという意見が多かったように記憶している．それに，「連係」ではなくて，「連繫」とかくべきであるとの強硬な意見もあった．しかし，当用漢字内で表現しようということから「連係」におさまった．しかし，この「連係編集プログラム」という表現は，このプログラムが本来もっている機能の微妙な部分をうまく表わした言葉だと思うのである．そもそも，連係という言葉にわれわれが一番馴染んでいるのは，スポーツにおいてである．野球とかサッカーのようなチームが協同で行うスポーツにおいて「連係プレー」がよいとか悪いとかという表現でしばしば使われる．最近では，企業内の人間組織の「連係プレー」が云々されることもある．連係といった場合，対象は１個ではなく複数個のものを前提としており，それらの間での有機的あるいは動的つながりが重要視されている．コンピューター・プログラムの場合にも，「連係」といった場合には，各サブルーチンやサブプログラム間での有機的つながりである．平た

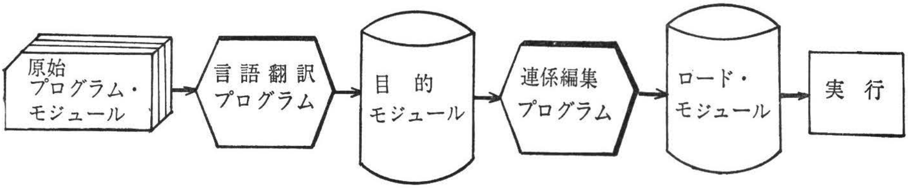
くいえば，サブルーチンやサブプログラム間での制御権の受け渡しやデータの受け渡しに関するつながりである．これら連係に関しては，「プログラム構造」の部分でもふれたように原始プログラムの作成過程での工夫が必要なことはいうまでもない．しかし，別々に言語翻訳された場合の相互の連係を実質的に解決するのは，連係編集プログラムである．さらに，このプログラムは，単にモジュール間の連係を解決するだけでなく，各々のモジュールに割り当てられている番地（アドレス）の調整，共通エリアの調整，不用モジュールの削除などといったいわゆる「編集」的機能を果している．ということからすると，この連係編集プログラムという名前は，その内実をよく表わしているといえるのである．

プログラム名の語源に紙面をとられすぎたが，これからこのプログラムの基本的機能を説明することから始めよう．

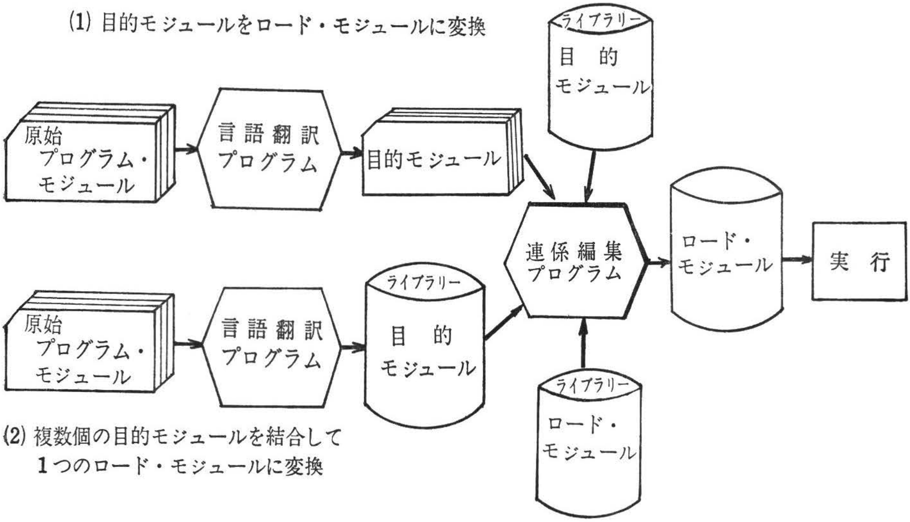
1.　連係編集プログラムの基本的機能

その第1機能は，すでに何回もふれたように目的モジュールをロード・モジュールに変換することにある．つまり，言語翻訳プログラムが出力した機械語のプログラムでありながら，いまだ実行不可能なプログラム・モジュールを実行可能なモジュールに変換する働きである（**第13図**(1)を参照のこと）．

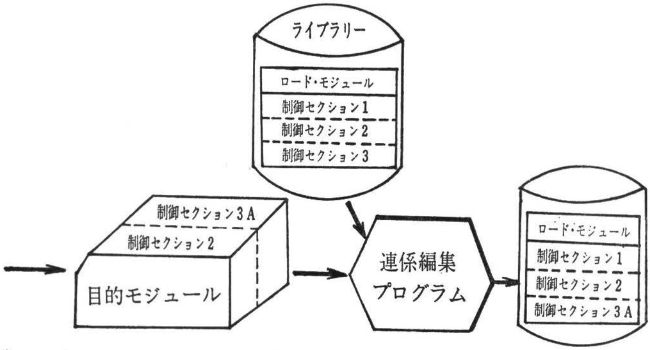
第2の機能は，モジュール間の結合という機能である．この機能を利用することによって，プログラマーは1つの問題をいくつかに分割してプログラミングができる．分割した各々の部分は，1人のプログラマーが書き上げることもできるし，何人かのプログラマーの分業で行ってもよい．プログラムを分割すれば，コーデングやデバッキングが容易になるし，それぞれの分割部分にふさわしいプログラム言語の選択も可能になる．分業すれば，それだけ短時間に1つのプログラムの完成が期待できる．ともあれ，連係編集プログラムは，それぞれ別々にコンパイルあるいはアセンブルされた複数個の目的モジュールを1つのロード・モジュールに仕立て上げ，出力する．出力モジュールは，**区分データ・セット**(「データ管理の諸機能とその構造」の章で詳述の予定）から成るライブラリーに収容される（**第13図**(2)を参照こと）．



(1) 目的モジュールをロード・モジュールに変換



(2) 複数個の目的モジュールを結合して
1つのロード・モジュールに変換



(3) ロード・モジュール内の特定制御セクションの置換

第13図　連係編集プログラムの機能

モジュールの連結にさいして，入力モジュールは，カード・デックや磁気テープ上に分散していてもさしつかえない．いくつかの入力源から入力モジュールを呼び込んできて，1つのロード・モジュールに組み込むこともできる．さらに，直接アクセス装置上にあらかじめ格納している標準的サブルーチンないしはサブプログラムを組み込んで1つのロード・モジュールを完成することもできる．直接アクセス装置上のサブプログラムやサブルーチンが収容される場所は，区分データ・セットの編成がなされており，一般にライブラリーと呼ばれる．このライブラリーからの呼び出し機能を活用すれば，プログラマーがそのつど共通的サブルーチンを書く手間がはぶける．連係編集プログラムの制御ステートメントを用いて，要求するサブルーチンを主プログラムの所要個所に組み込むように指定すればよい．連係編集プログラムは，入力データの中にこのような制御ステートメントがあると，そのステートメント上に指定されているライブラリーから該当するサブルーチン（目的モジュールであってもロードモジュールであってもかまわないが）を捜し出して，出力ロード・モジュールに組入れてくれる．

入力モジュールの処理が終っても，まだ未解決の外部記号，つまり，別のモジュールを参照している記号が残っている場合には，**自動的ライブラリー呼び出し機能**（automatic library call）が働く．この機能の働きで，未解決の記号に合致するサブルーチンを特定のライブラリーから捜し出してきて，出力モジュールの1部として組み込んでくれる．この機能が活用されるので，たとえば，FORTRAN 言語で書いたプログラムに，言語翻訳された段階ではFORTRAN 用のサブルーチンが一切組込まれていなくても，連係編集プログラムの実行段階で FORTRAN のサブルーチン・ライブラリーから必要なサブルーチンが続々と自動的に呼び出されてきて完結したプログラムとなるのである．言語翻訳の段階でいちいちサブルーチンを呼び出してきてコンパイルしていたのでは時間がかかってしまう．連係編集プログラムのステップで自動的にサブルーチンを呼び入れた方がはるかに得策であることは容易に理解できよう．

さて，ここで**ライブラリー**について若干ふれておこう．目的モジュールを収容するライブラリーは，オペレーティング・システムを利用するユーザが任意

に設定することができる．したがって，目的モジュール・ライブラリーに関しては特に言及しないでおこう．一方，ロード・モジュールを収めるライブラリーには，３つのタイプが考えられる．「一時的ライブラリー」,「私用ライブラリー」,「システム・ライブラリー」の３種類である．

一時的ライブラリー（temporary library）は，連係編集プログラムが出力したロード・モジュールを同一ジョブ内の後続ジョブ・ステップで実行することを前提として，一時的にロード・モジュールを貯える区分データ・セットである．したがって，一発主義的なプログラムの収容とか，テスト段階のプログラムを収容するのが主目的である．

私用ライブラリー（private library）は，使用頻度はあまり高くないが，２回以上にわたって繰返し使われるロード・モジュールを貯えるのに使われるものである．一時的ライブラリーの中のロード・モジュールが，連係編集プログラムの実行後の同一ジョブ内のステップでしか有効でないのに対して，私用ライブラリーに収容したロード・モジュール群は，長期的に有効である．

システム・ライブラリー（system library）は，使用頻度の高いプログラム（ロード・モジュール）を保管するための区分データ・セットである．SYS1. LINKLIB とか SYS 1. SVCLIB とかの固有のデータ・セット名がつけられれている．実は，連係編集プログラムとか言語翻訳プログラム自身は，SYS1. LNKLIB に収容されている．SYS 1. SVCLIB には，非常駐の制御プログラムの大半が収容されている．なお，システム・ライブラリーに，ユーザが作成したプログラムを収容することも一向に差しつかえない．ともあれ，連係編集プログラムで出力されるロード・モジュールは，これらライブラリーのいずれかに貯えられることになる．

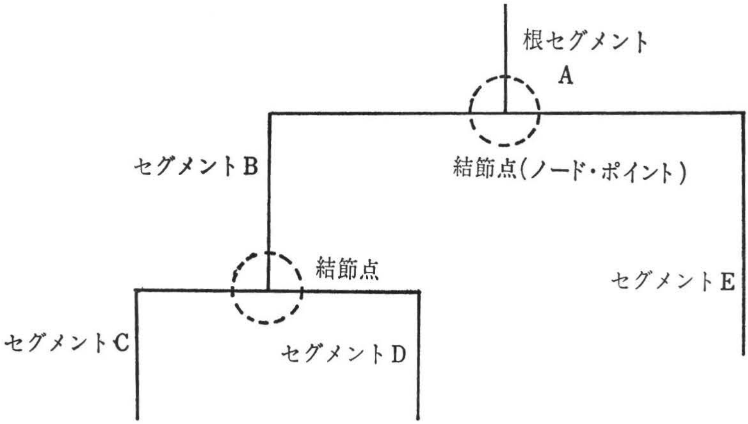
話をもとにもどそう．連係編集プログラムの主要な機能として，実行可能なロード・モジュールの作成とモジュール間の結合があることについてはこれまでにふれた．モジュール間の結合にあたっては，別々にコンパイルまたはアセンブルした目的モジュールを１つのロード・モジュールにまとめ上げることも出来るし別の入力源から目的モジュールやロード・モジュールを呼込んで結合することもできた．さらに，未解決の記号が残った場合には，自動ライブラリ

ー呼び出しの機能が働く．

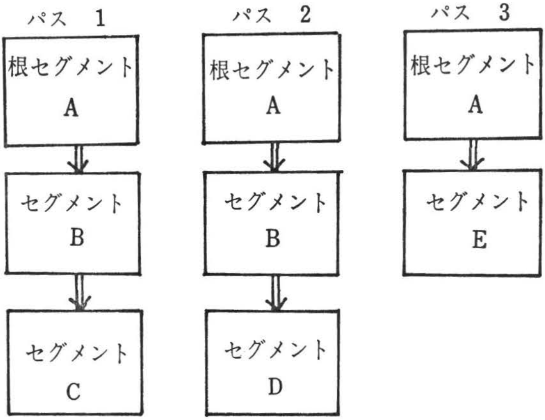
では，第3の機能に移ろう．**第13図の(3)**を参照していただきたい．プログラム修正の機能である．連係編集プログラムがもっている“編集”の機能を用いることで容易にプログラムの修正や変更ができる．プログラム全体を言語翻訳し直すのではなくて，変更や修正の影響する**制御セクション**(control section)だけに手を加えたうえで，元のプログラムの一部分を置き変えることができる．制御セクションとは，それ自体で1つのまとまったコーディングの単位である．プログラム作成における最少単位は，この制御セクションである．つまり，原始モジュールは，1つ以上の制御セクションからなり立つ．また，連係編集プログラムへの入力データである目的モジュールの最少単位も制御セクションである．さらに，プログラムの実行にさいしては，プログラム・ローディングや再配置の最少単位でもある．このような意義をもつ制御セクションが，連係編集プログラムの行なうプログラムの変更・修正の単位となるのだ．連係編集プログラム用の制御ステートメントを使って，すでにライブラリーに登録されているロード・モジュール内の特定制御セクションだけを別の修正した制御セクションで置きかえたり，削除したり　あるいは，別の位置に配置がえしたりが可能なのである．場合によっては，外部を参照する記号を制御ステートメントを使って変更したり削除したりすることもある．また，制御ステートメントの指示によらずに，連係編集プログラムに自動的に特定制御セクションを置きかえさせることだって可能である．

連係編集プログラムの見落すことのできない第4の機能は，計画的オーバレイ構造を作り上げることである．計画的オーバレイの基本的考え方についてはすでに「計画的オーバレイ構造と監視プログラムの働き」の項で説明しておいたが，ここでは，連係編集プログラムにかかわりをもつ部分だけをとりあげて説明しておこう．

あるプログラムを実行するにあたって，主記憶装置という資源を最少限に，かつ有効に使うために用いられる手法が，オーバレイであることはいうまでもない．しかし，同じオーバレイを行わせるにしてもあらかじめプログラムの実行順序が整然と判別し得ないとか，その予測が立てにくいほど入りくんで複雑



樹　木　式　構　造



セグメント間の依存関係

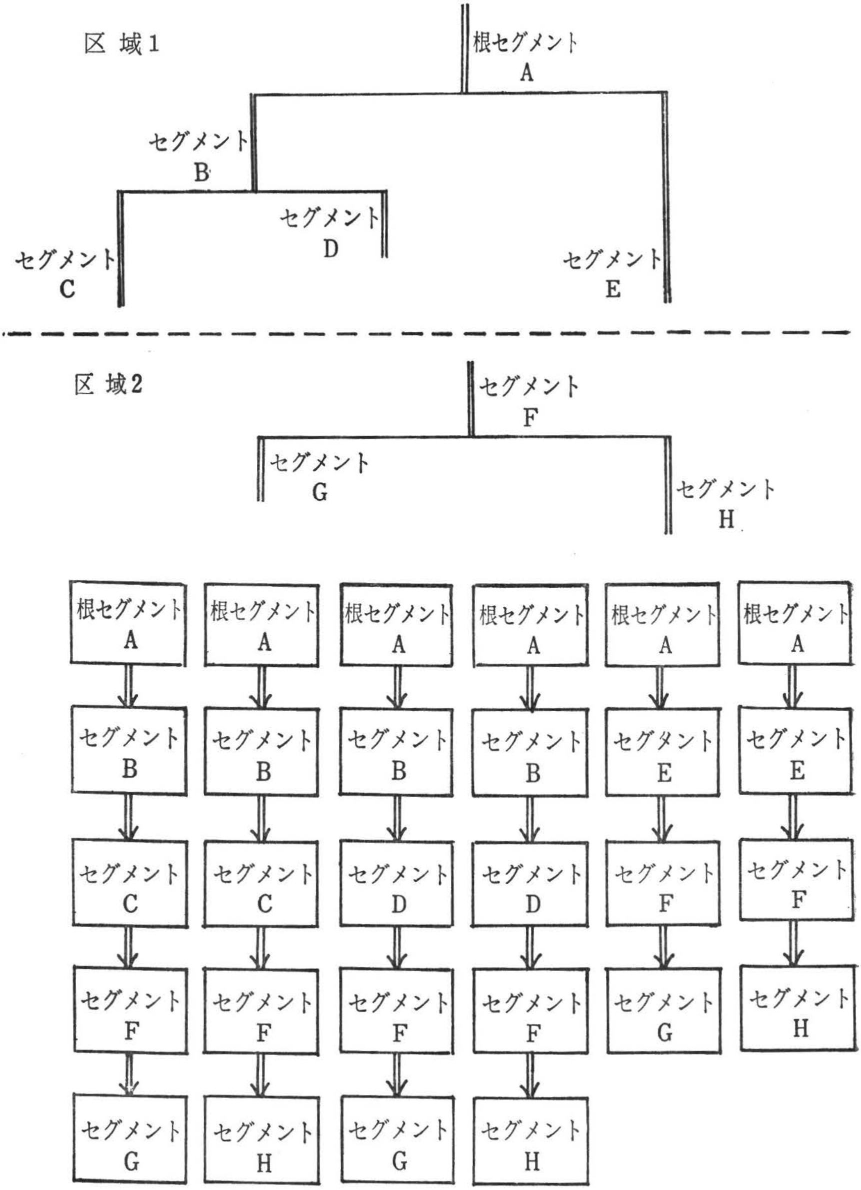
第14図　　オーバレイの樹木式構造と依存関係

な実行順序をとる場合には動的なオーバレイ手法を採用するしか手がない．しかし，どの部分をどのようにオーバレイすべきかの計画があらかじめ容易に行いうる場合には，計画的オーバレイを使う．プログラマーは，解決すべき問題の処理内容の相互関係にもとづいて，プログラムをいくつかのセグメントに分割することを想定してプログラムの設計を行えばよい．これが，オーバレイの樹木式構造の考え方である（**第14図参照**）．この図における**セグメント**（segm-

ent)とは，プログラムの実行中に，１つのまとまった論理的単位として主記憶装置にローディングされる最少単位である．したがって，セグメントは，最低１個以上の制御セクションから構成されることになる．なお，**根セグメント**（root segment）だけは，プログラムの実行中常に主記憶装置に貯えられるべき性質のセグメントである．他のセグメントは，それが必要になった時点で主記憶装置にローディングされていないときは，必ずオーバレイの方法で呼び込まれる性質のものである．

樹木式構造（tree structure）は，何種類かの一連のセグメントの組合わせが主記憶装置をどのような形式で占有するかを図式化したものである．このような樹木式構造を作るにあたっては，同時に主記憶装置内に入っている必要のない２つあるいはそれ以上のセグメントを相対立させるように作図してゆく必要がある．第14図において，セグメントＢとＥ，ＣとＤは同時点で主記憶装置に共存する必要のないことを表わしている．さらに，ＣとＤがＥと共存する必要もない．このオーバレイの樹木式構造にもとづいて，“OVERLAY” 制御ステートメントを用いて，各セグメント間の相互関係を詳細に指定してやればよい．このさい，セグメント間の相互関係を示すのに使われるのが**結節点**（node point）である．連係編集プログラムは，制御ステートメントの指示に従って，同時に主記憶装置内に入っている必要のない２つ以上のセグメントに対しては，同じ相対的アドレスを重複して割り当てる．その結果，それらセグメントが同じ記憶域にオーバレイの形式でロードされるようになる．なお，ロード・モジュールを構成する複数個のセグメントは，プログラムの実行にさいして制御プログラムがそれらを別々に読み込むことができるような形式でライブラリーに登録される．

第14図の下側に示したのは，上側の樹木式構造に含まれる一連のセグメントで，末端のセグメントと根セグメントを結ぶ経路を構成する複数個のセグメントの相互関係である．これらのことを一般に**パス**（path）と呼んでいる．このパスにおいて，セグメントＣがロードされているときには，必ずＡとＢもロードされていることを意味している．セグメントＤの場合も同様である．根セグメントＡは，何時でもこのプログラムが実行されているかぎり記憶装置内に常



第15図　　複数個の区域におけるオーバレイの樹木式構造

駐していることを意味している．

第15図の場合は，２つの区域を使う場合の樹木式構造である．ここでいう**区域**（region）とは，主記憶装置の連続した記憶域のことであり，他の区域内のパスとは独立して，その中にいくつかのセグメントを連続して読み込むことのできる記憶域である．したがって，ある同一区域内のパスとしては，同一時点では，ただ１つしか主記憶装置に存在できない．第15図に示したように，オーバレイ構造をとるプログラムを２つ以上の区域に入れるように設計することもできる．２つの区域は，連続した区域である必要はない．制御プログラムは，それぞれの区域として，独立した２つの記憶域を見出して割り当ててくれるので，記憶域の有効利用が計れる．そればかりではない．複数個の区域をもつオーバレイ構造では，一方の区域のセグメントから，それ自身のパスにはないセグメントを指すことができる．また，複数個の区域を用いる構造では，セグメントを読み込む効率を高めるという利点もある．というのは，次に実行すべきパスをある区域に読み込んでいる最中に，併行して別の区域では処理を続行することができるからである．

ともあれ，計画的オーバレイ構造をとるための出発点は，まず“樹木式構造”をしっかりとしておくことである．この樹木式構造を作るに当っては，使用可能な主記憶装置の容量，各制御セクションの使用頻度と相互の依存ないしは従属関係，１つのパスの内部での制御権の移動の仕方あるいは，あるパスから別のパスへ，またはある区域から別の区域への制御権の移動の仕方を慎重に検討する必要がある．構造が決定したら，その構造にしたがって連係編集プログラム用の制御ステートメントで指定すればよい．単一区域にしろ，複数区域にしろ，それに見合ったオーバレイ構造のロード・モジュールを完成する仕事は連係編集プログラムが行ってくれる．ただし，あるセグメントから別のセグメントを CALL する命令は，当然のことながら，もとのセグメントに入っている必要がある．それについては，ここではふれない．

連係編集プログラムの主要な機能は，以上の４つであるが，これらのほかに**共通域**（common area）の処理，ロード・モジュールに対する別名の設定あるいは**入口点**（entry point）の指定ないしは変更を行ってくれる．これらの

機能について簡単にふれておこう.

アセンブラー言語や FORTRAN 言語では, COM あるいは COMMON ステートメントの指定によって, 共通記憶域を確保することができる. これらステートメントで確保した共通域は, サブプログラム（またはモジュール）間で共用するためのものである. PL/I 言語で確保する**静的外部記憶域**（static external storage area）もこれに相当する. 別々に言語翻訳した目的モジュールにこの共通域がそれぞれのモジュールに散在しているのであるが, 連係編集プログラムは, これら共通域を1つの個所にまとめ上げ, 出力ロード・モジュールの中に正しく位置づけてくれる.

ロード・モジュールには, 必ずそのモジュールを呼びだすために名前をつけておく必要がある. ロード・モジュールをローディングして, 実行を行うにはこの名前が使われる. ロード・モジュールに名前をつける働きは, 連係編集プログラムにある. 連係編集プログラム用制御ステートメントである NAME を使って, 本名をつけることができる. 本名以外に別名をつけるには, ALIAS ステートメントを使う. 5つまでの別名がつけられる. ロード・モジュールを呼び出すには, 本名を使うこともできるし別名を使ってもよい. 別名を使うことによって, プログラムを実行する最初の命令, つまり入口点を変えることが可能になる.

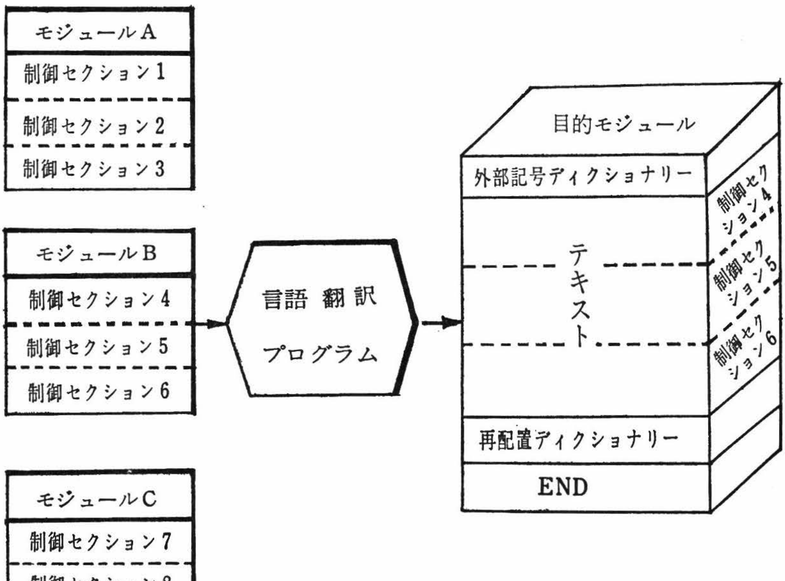
通常, プログラムの入口点は, 原始プログラム作成時に指定される. たとえば, アセンブラー言語の場合だと, END ステートメントのオペランドで指定した記号が入口点をさす. コンパイラーの場合だと, 実行可能なステートメントの最初のものが入口点となる場合が多い. ところでいくつかの目的モジュールを結合して, 一つのロード・モジュールとした場合の入口点は, 連係編集プログラムで指定できる. ENTRY 制御ステートメントを用いて最初に実行すべき命令（入口点）の指示をすることによって, 連係編集プログラムに処理をさせる. 原始プログラム段階で指示されている入口点は, この ENTRY ステートメントで指定された入口点で置き換えられることになる.

2. 目的モジュールとロード・モジュールの構造

連係編集プログラムの機能を構造的に把握するには，連係編集プログラムの入力源となる目的モジュールとロード・モジュール，およびその出力モジュールであるロード・モジュールがどのような構造をもっているかの理解がないといけない．そのために，ここでは，目的モジュールとロード・モジュールのモジュール構造について言及しておくことにしよう．

どのような問題プログラムにしろ，特定の問題解決を目的として作成されることはいうまでもない．その目的を達成するために，複雑な問題の場合だと，プログラムは，それぞれの特定機能を行ういくつかの論理的単位に分割してコーディングするのが普通である．ある特定の機能，あるいは互に関連性の深いいくつかの機能を果すコーディング上の論理的単位が“モジュール”の出発点である．モジュール化（積木方式）の意義については，本章の最初でくわしく説明した通りである．

第16図に示したように，１つのモジュールは，１つあるいは２つ以上の制御



第 16 図　プログラムを構成する原始モジュールの制御セクションおよび目的モジュールの構成

セクションから構成される．プログラマーは，１つのモジュールを複数個の制御セクションに分割することもできるわけだ．その場合，各々の制御セクションが主記憶装置のどこにローディングされてもいいように，一つのセクションから別のセクションへの制御権の受け渡しが正しく行われるための配慮が必要である．すでにふれたように制御セクションは，それ自体で１つにまとまったコーディングの一単位であり，実行にさいしてのローディング時には，他の制御セクションとは独立して再配置が可能である．FORTRAN 言語や COBOL 言語などのような高級言語でプログラムの作成を行う場合には，何ら制御セクションに関しての配慮はいらない．アセンブラー言語を使用する場合には，この制御セクションを意識する必要が生じる．アセンブラー言語においては，CS-ECT 命令で制御セクションを定義するのが普通である．最初の制御セクションだけは，START 命令で定義する．アセンブラーにとってみれば，プログラムのようなものは何も存在しない．その代り，１つ以上の制御セクションの集まりこそが，アセンブラーに対する入力データとしての意味をもっている．また連係編集プログラムにとっても，ロード・モジュールを形式するための最少単位は制御セクションである．たとえ，高級言語で作成したプログラムであっても，目的モジュールやロード・モジュールを構成する基本単位は，制御セクションであることに変りない．

第16図で示してあるように，言語翻訳プログラムの出力である目的モジュールは，制御セクションの集合体であるテキストの部分と，制御目的のディクショナリーから構成されている．**制御用ディクショナリー**（control dictionary）には，目的モジュールを結合するさいに制御セクション間やモジュールの間での相互参照を解決するのに必要となる情報が収容されている．**テキスト**（text）には，機械語の命令とデータが入っている．つまり，機械語プログラムの部分である．

まず，外部記号ディクショナリーから説明してゆこう．**外部記号ディクショナリー**は，External Symbol Dictionary という言葉からきており，一般に ESD と略される．

ESD には，そのモジュール以外のモジュールから参照することのできる記

号名と，逆にそのモヂュールから別のモヂュールを参照するための記号名が集められている．前者が**外部名**（external name）と呼ばれるのに対して，後者は**外部参照**（external reference）と呼ばれる．ESDを構成する各々の登録項目には，1つの記号名とその記号名がそのモジュールの中のどこで使われているか（つまり，その記号名が使われている番地）の情報が入っている．

“外部名”は，先述したようにどの制御セクションからでも，あるいは，別個に言語翻訳されたモジュールからでも指定することのできる名前である．ESDの中では，おのおのの外部名とその外部名が該当モジュールの中のどこに位置しているかの番地が示されている．したがって，連係編集プログラムで2つのモジュールを結合する時，一方のモジュールのESDにある“外部参照”のための記号を別のモジュールの中にある“外部名”とつき合わせ，両者の記号名が一致したとき，外部名に対応する番地を取り出して，“外部参照”記号名の番地とするのである．こうすることによって，一方のモジュールから別のモジュールにあるサブルーチンに制御権を移したり，データ項目を受け渡したりすることが可能になるわけだ．外部名として登録されるものとしては，制御セクション名，入口名あるいは共通記憶域名などがある．制御セクション名は，CSECT命令またはSTART命令で定義される．入口名は，ENTRY命令で指定されるものであり，共通記憶域は，COM命令で指定される．

一方，“外部参照”名は，該当モジュールとは別に言語翻訳された目的モジュールの中で外部名として定義されているべきものであり，該当モジュールでは別のモジュールの中にあるルーチンやデータを参照する目的で使われている記号である．この記号名は，アセンブラー言語では，EXTRN命令やVタイプのアドレス定数を使って指定されるものである．

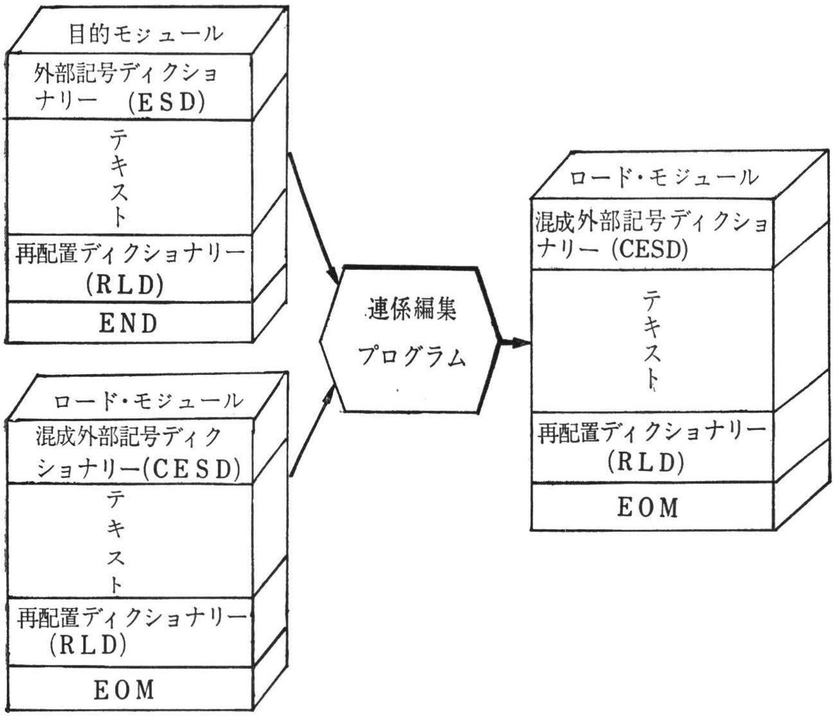
テキストには，該当モジュール内の命令とデータとが入っている．命令とデータおよびモジュール内でそれらに割り当てられた番地が1つのテキスト項目を形式する．テキストの部分が，いわゆる機械語プログラムの部分に相当する．**第16図**に示したごとくテキストの部分は，いくつかの制御セクションから成り立っている．

再配置ディクショナリーは，Relocation Dictionary という言葉からきてお

り，略して RLD といわれる.

この RLD には，該当モジュール内で使われている**アドレス定数**（address constant）に関する情報が入っている．アドレス定数は，そのモジュールが再配置されることによって修正を余儀なくされる番地である．そのモジュールがたとえば，2000番地からローディングされるとすると，おのおののアドレス定数を2000だけ増すことによって，プログラムの再配置が調整されるわけだ．したがって，RLD の各登録項目には，それぞれのアドレス定数が該当モジュール内のどこで使われているかが示されている．プログラムのローディング時に各項目を参照して，再配置にともなうアドレス調整ができるわけである．

目的モジュールの最後のレコードは，END ステートメントである．原始プログラム・モジュールの END ステートメントに対応するのがこれである．



第17図　連係編集プログラムによって作り出されるロード・モジュール構造

連係編集プログラムの処理結果であるロード・モジュールの構造を示したのが**第17図**である．ロード・モジュールは，目的モジュールの基本的構造と変りはない．しかし，目的モジュールが単に再配置可能な形式をとっているのに対して，ロード・モジュールは再配置可能でかつ，実行可能な形式である．ロード・モジュールは，連係編集プログラムへの入力として用いられたすべてのモジュールの組合わせから構成される．その入力モジュールは，第17図に示したように目的モジュールとロード・モジュールであっても差しつかえない．両者は，一定の規則に従って組合わされ1つの実行可能なロード・モジュールの形態となる．テキストの部分は，両者が一緒になって1つのテキストを形作る．ESD の部分は，両モジュールの外部名と外部参照がつき合わされ，相互の未解決記号の番地を決定したうえで1つに**統合された外部記号ディクショナリー**(Composite External Symbol Dictionary, 略して CESD) を構成することになる．これによって，モジュール間での制御権の受け渡しやデータの参照に関する問題の解決がなされるのである．また，両モジュールが有している再配置ディクショナリーも統合されて，1つの RLD となる．さらに，ロード・モジュールの最後のレコードとしては，END の代りに EOM (End of Module) レコードが作り出され，モジュールの終りを示すようになる．このようにして連係編集プログラムは，複数個のモジュールをあたかも1つのモジュールとして言語翻訳したかのごとく，見事に1つのロード・モジュールとして仕立て上げてくれるのである．連係編集プログラムは，つぎのような処理を行うことによって実行可能なロード・モジュールを作成してくれるといえる．まず第1に出力モジュールを構成する制御セクションに対して相対的な主記憶装置の番地を割りふる．それぞれの入力モジュールは，別々に言語翻訳プログラムで割り当てられた番地をもっているがゆえに，入力モジュールの番地は，重複している可能性がある．たとえば，2つのモジュールが同一起点にもとづいてアセンブルされているとすると，全く同一記憶域が割り当てられていることになる．そこで，連係編集プログラムは，最初の制御セクションに起点を割りふったら，この起点を基準に各々の命令やデータの番地を割り当てる．ひきつづいて，つぎの制御セクションの起点を決め，そのセクション内の命令やデータには，そ

のセクションの起点に割り振った番地だけ再配置した番地を与えてゆく．このような方法で，相互に番地が調整され，テキストの完成をみるのである．また入力モジュール間の外部参照に関しての解決処理を行ってくれることは，いうまでもない．異ったモジュール間での相互参照は，記号を目印として行っているので，出力ロード・モジュールに割り当てられた連続する主記憶装置の番地の相対的位置にもとづいて解決される．この記号による相互参照は，アドレス定数を媒介として行なわれている．したがって，連係編集プログラムは，再配置によって移動したバイト数だけアドレス定数を調整したうえで，そのアドレス定数を参照している外部参照記号の番地を指定することによって相互参照の解決をする．このような処理を行うことによって，複数個のモジュールがあたかも１つのモジュールとして言語翻訳されたと同じような形式のロード・モジュールを作り上げるのである．

なお，連係編集プログラムによる出力モジュールには，そのモジュールを実行するさいに主記憶装置にローディングし，再配置するのに必要なすべての情報が貯えられている．連係編集プログラムが，ロード・モジュールを仕立て上げてライブラリーに登録するさいに，そのモジュールに対する**登録簿**（directory）が作られる．この登録簿には，モジュールの本名や別名以外に，ロード・モジュールの構造や内容あるいは形式に関する情報が同時に登録される．これら情報のことを一般に**モジュールの属性**と呼んでいる．モジュールの属性を示す情報は，制御プログラムがプログラム・ローディングにさいして利用するものである．目的モジュールには，モジュール属性がないので，制御プログラムでローディングして実行させることができないのである．属性のうちのあるものは，プログラマーが制御ステートメントを使って指示できる．また，あるものは，連係編集プログラムが，処理過程で収集した情報にもとづいて作り出すものがある．プログラマーが制御ステートメントで指定できるものとして，モジュールの構造を示すための「再入可能」，「逐次再使用可能」などがある．また，形式を示すものとして「オーバレイ形式」とか「分散形式」などがある．さらに，内容を示すものとしては「テスト」とか「編集不能」などが指定できる．一方，連係編集プログラムが作り出す属性としては，「ブロック形

式」，「実行可能」，「実行不能」などがある．

これら属性に関する情報を手がかりとして，制御プログラムは，そのモジュールのローディングの仕方を決めるのであるが，その詳細はここでは省略する．

3. 連係編集プログラムの入出力構造とその指定

連係編集プログラムの入力データとしては，目的モジュールと連係編集用制御ステートメント，それにロード・モジュールが考えられる．すでに説明したような連係編集プログラムのさまざまな機能のうちのいずれかを実現させるには，当然のことながらそれを要求するための指示を行う必要があろう．そのために使われるのが連係編集プログラム用制御ステートメントである．ただし，連係編集プログラムの特別な機能，たとえば，診断用出力データに関する選択だとか，出力ロード・モジュールにかかわりをもつ属性については，ジョブ制御ステートメントの1つである EXEC ステートメントのオペランドで，パラメーターとして指定できるものもある．ともあれ，連係編集用制御ステートメントは，目的モジュールと一緒に，あるいは，それだけを1まとめにして入力することができる．

目的モジュールは，原始プログラムを言語翻訳して直ちに連係編集するために入力する場合もあるが，あらかじめ言語翻訳プログラムを通して目的モジュールを作っておき，他のモジュールと結合するために入力することもある．特定モジュールとだけ結合して使うようなものは，磁気テープや磁気ディスク内に順次データ・セットとして保管する方法がとられよう．あるいは，目的モジュール・デックとして保管しておき，他のモジュールが完成したとき，一緒にして連係編集プログラムに入力すればよい．ところで，目的モジュールでも，いくつかの異ったモジュールで結合使用される可能性のあるもの，つまり使用頻度の高いサブルーチン的なものは，あらかじめサブルーチン・ライブラリー（一般には区分データ・セット）に登録しておく方法がとられる．そのサブルーチンを必要とするモジュールが連係編集プログラムに入力されたとき，ライブラリーからそのサブルーチンを呼び出して組み込む方式をとった方が得策で

ある.

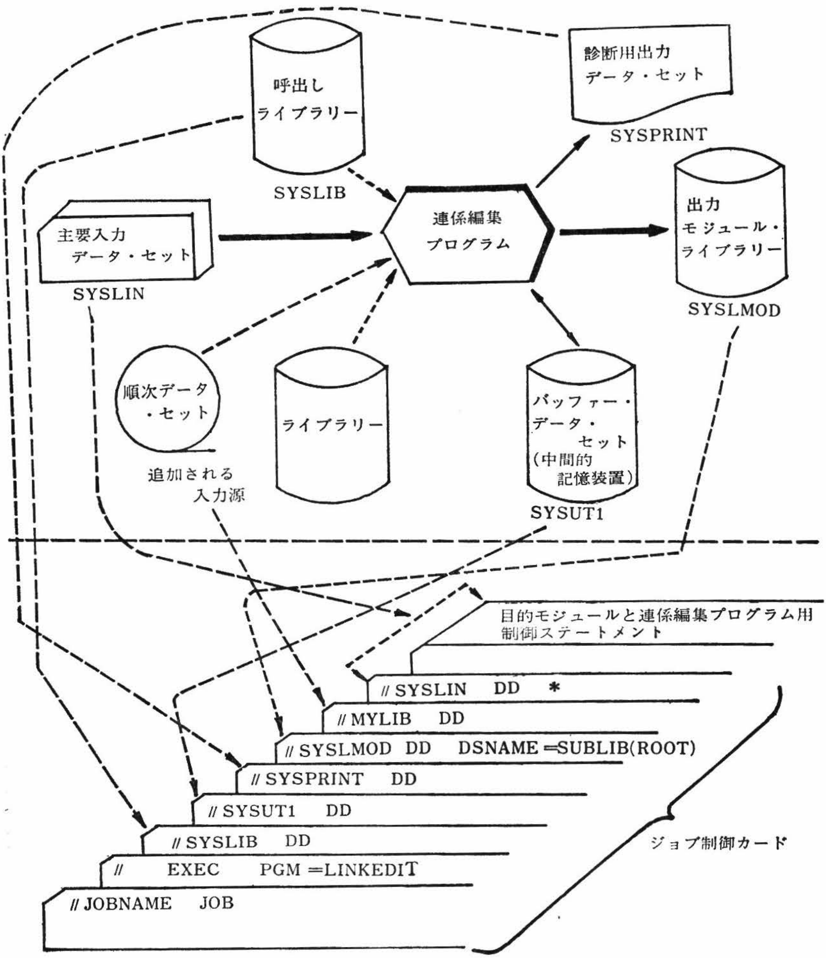
さて，連係編集プログラムへの第3番目の入力データとして使えるのは，ロード・モジュールである．ロード・モジュールを連係編集プログラムへの入力として適用する主目的としては，つぎの2つの場合が考えられる．ロード・モジュールのサブルーチン・ライブラリーを作っておいて，連係編集プログラムの実行段階で，他のモジュールにサブルーチンを組み込む場合がある．別の場合は，完成後のロード・モジュールの一部分の手直しを効率よく行う場合である．手直しの必要のある制御セクションだけを訂正し，言語翻訳したあとで，連係編集プログラムの機能を利用することによって，元のロード・モジュールの該当制御セクションだけを自動的に置き換えるときに，ロード・モジュールが入力として使われる.

単一目的モジュールだけの連係編集，複数個の目的モジュールを同一入力媒体から入力しての連係編集，ある媒体から入力した目的モジュールとすでにライブラリーに収容されているサブルーチン的目的モジュールの連係編集，あるいは，目的モジュールとサブルーチン的ロード・モジュールの連係編集等々，多彩な入力モジュールの結合が想定されるとしたら，当然，連係編集プログラムの入力源としてさまざまなものが考慮される必要がある.

1）さまざまな入力源

連係編集プログラムは，主要入力データ・セット，自動呼出しライブラリー，その他の追加データ・セットから入力データを受け入れることができる（**第18図参照**).

主要入力データ・セット (primary input data set) は，連係編集プログラムを使用する場合には，必ず必要となるデータ・セットである．オペレーティング・システムでいうデータ・セットとは，一群のデータの集まりのことで，そのデータ群を総称する名前，つまりデータ・セット名をもったものである．一般にはデータ・ファイルと同義であるが，特定の名前がつけられ，その名前を使ってファイルが制御管理されるという点にデータ・セットの特異性がある．ともあれ，主要入力データ・セットは，連係編集プログラムにとって必須



第18図　連係編集プログラムの入出力源とジョブ制御カードの関係

のデータ・セットであり，これを使って制御ステートメントと目的モジュールが入力される．場合によっては，このデータ・セットからは制御ステートメントだけを入力し，目的モジュールは他のデータ・セットから入力することもある．主要データ・セットは，順次編成のデータ・セットである．なお，**主要入力データ・セットのデータ定義名**（Data Definition name, 略して DD 名）は SYSLIN である．DD 名は，先述したデータ・セット名（略して DS 名）

とは異る．DS 名はデータ・セットを参照するのに使われるものであるが，DD 名はジョブ制御カードの1つである DD ステートメントに用いられる名前であり，プログラムの中で記述されている**データ制御ブロック**（Data Control Block：略して DCB）との関連性を示すのに使われる．データ制御ブロックとはデータ・セットに関する情報，つまりり，データ・セットを構成するレコードやブロックの長さ，レコードの形式およびデータ・セットの編成方法（順次データ・セット，索引順次データ・セット，直接アクセス・データ・セット，区分データ・セット）などの情報が集約されている記憶域である．このデータ制御ブロックと呼ばれるものとジョブ制御ステートメントの1つであるDDステートメントの中の該当するものを結びつけるのがDD名というわけだ．プログラムの中で記述されたデータ制御ブロック内の不足する情報を，そのデータ制御ブロックに対応する DD ステートメントで補うことができるのである．つまりプログラムの実行段階でデータ・セットのさまざまな属性を与えることができるようになる．データ・セット，データ・セット名，データ定義名，データ制御ブロックなどと耳なれぬ言葉が続出したが，くわしくは「データ管理の諸機能とその構造」の章で説明するので，ここでは深入りしない．

話をもとにもどそう．主要入力データ・セットは，連係編集プログラムにとって不可欠であり，連係編集用制御ステートメントと目的モジュールの入力として使われるということが重要なのである．

呼出しライブラリー（call library）は，自動的ライブラリー呼出し機能を利用する場合だけに必要となる．自動的ライブラリー呼出し機能は，入力モジュールがすべて処理されたあとに，まだ解決されない外部参照用の記号が残っているさいに働き，その記号をもっているモジュール（サブルーチン）を探し出し，出力モジュールの1部として組み込む機能であった．通常の場合，この自動呼出し機能は自動的に働くのであるが，ジョブ制御ステートメントのうちのEXEC ステートメントのオペランドでパラメータとして NCAL と指定した場合にだけ働かなくなる．自動的呼出しライブラリー機能を使用するときには，あらかじめそのライブラリーには必要となるサブルーチンが登録されている必要がある．呼出しライブラリーには，目的モジュールの形式でサブルーチ

ンを入れておいてもよいし，ロード・モジールの形式で登録してあってもよい．ただし，同一ライブラリーに両者が混在することは許されない．目的モジュールだけで構成されるライブラリーかロード・モジュールだけで構成されるライブラリーのどちらかである．目的モジュールである場合には，そのライブラリーには，目的モジュールと連係編集用制御ステートメントが入っていてもよい．しかし，ロード・モジュール・ライブラリーの場合だと，ロード・モジュールだけしか登録できない．呼出しライブラリーは，区分編成のデータ・セットであり必ず直接アクセス装置に収容される必要がある．**区分データ・セット**とは，一般に**プログラム・ライブラリー**と呼ばれるものであり，収容されるモジュールごとに独立した区域が与えられ，それぞれのモジュールがどの区域に収容されているかは登録簿（ディレクトリー）で示されるような形式のデータ・セットのことである．なお，呼出しライブラリーのデータ定義名は SYSLIB である．

主要入力データ・セットと呼出しライブラリー以外にも自由に入力源を追加することができる．追加される入力源は，順次データ・セットでも区分データ・セットでもかまわない．順次データ・セットの場合だと，目的モジュールの形式しか取り扱うことはできないが，区分データ・セットの場合には，目的モジュールあるいはロード・モジュールのいずれであってもかまわない．これら追加される入力源のデータ定義名は，利用者が自由につけることができ，各々データ定義ステートメントで定義しなければならない．

追加の入力源から所要のモジュールを出力ロード・モジュールに組み込むには INCLUDE ステートメントを使用する．このステートメントのオペランドで指示されたデータ・セットの中から該当モジュールが，その場で組み込まれる．追加の入力源が区分データ・セットのときにかぎり，LIBRARY ステートメントも使える．このステートメントを使うことによって，呼出しライブラリー以外のライブラリーから自動呼出しを可能とすることができるのである．INCLUDE ステートメントでは，該当モジュールがその場に組み込まれるのであるが，LIBRARY ステートメントの場合だと，すべての入力モジュールが処理されたあと，このステートメントのオペランドで指示された名前がなおかつ

未解決のときに該当するモジュールが出力ロード・モジュールに組み込まれるのである．

以上のように，主要入力データ・セット，呼出しライブラリーおよび任意の追加入力源が，連係編集プログラムの入力として使えるのである．**第18図**を再度参照していただきたい．それぞれの入力源は，データ・セットに対応するDD ステートメントを用いてその性質を定義する必要がある．DD ステートメントでどのように定義するかについてはここではふれない．「ジョブ管理の諸機能とその構造」の章で詳説する．ここでは，DD ステートメントのオペランドで，データ・セットの性質や処理に関するさまざまな情報が定義可能であるということだけ理解していただいておけばよい．

2）中間記憶と出力データ・セット

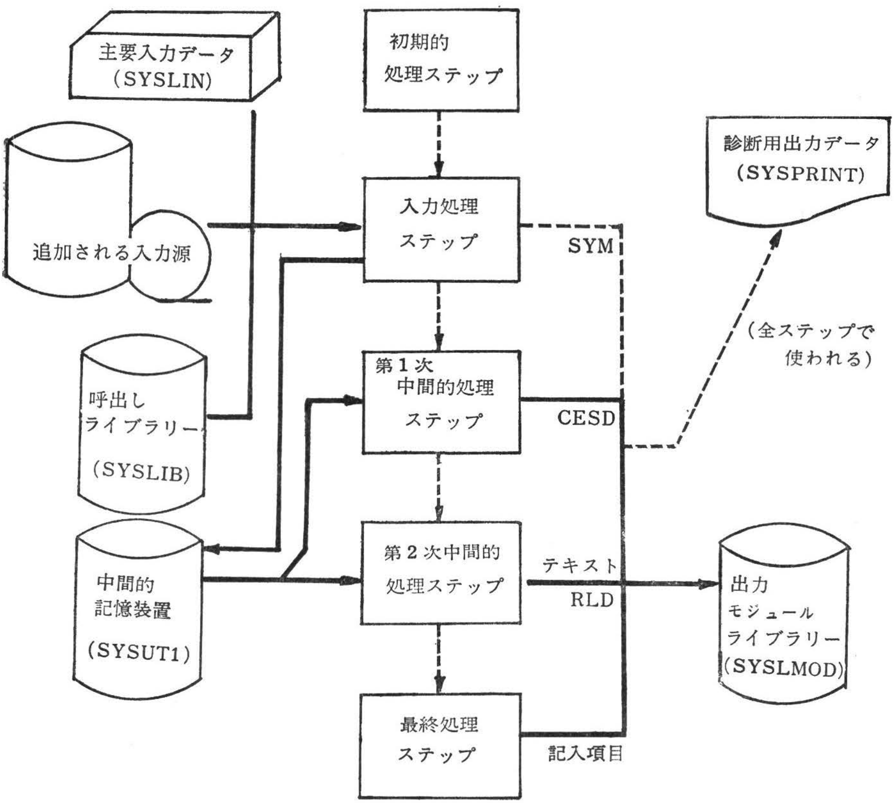
連係編集プログラムが処理を行っている過程で，主記憶装置に収容しきれないような中間的なデータは，**バッファー・データ・セット**（buffer data set）に一時的に貯える．バッファー・データ・セットは，いわばワーク・ファイルである．このバッファー・データ・セットは**中間記憶装置**とも呼ばれるものであり，直接アクセス装置であることが要求される．もちろん，呼出しライブラリーや追加ライブラリーなどと同一装置上の空いた場所であっても一向に差しつかえない．このデータ・セットのデータ定義名は，第18図にも示しているようにSYSUT 1 であり，必ず DD ステートメントによって定義しておく必要がある．

連係編集プログラムが作り出したロード・モジュールは，**出力モジュール・ライブラリー**（output module library）に登録される．このライブラリーは，いうまでもなく区分データ・セットであり直接アクセス装置上に作られる．ライブラリーである以上多数のロード・モジュールが登録可能である．それぞれのロード・モジュールはそのライブラリーのメンバーとして名前とともに登録される．ロード・モジュールそのものは，区分データ・セット内の特定区分内に収容され，名前とロード・モジュールがある場所を示す情報は登録簿に記録されるわけだ．

この出力モジュール・ライブラリーも，DD ステートメントで定義する必要がある．DD 名は，SYSLMOD である．この DD ステートメントのオペランドで出力ロード・モジュールの名前（一般には**メンバー名**いうのだが）を指定することもできる．第18図では，ROOT がそれにあたる．SUBLIB というのは，データ・セット名である．もし，メンバー名をこのステートメントの中で指定しない場合には，連係編集用制御ステートメントの１つである NAME ステートメントを使って指示する必要がある．DD ステートメントと NAME ステートメントの両方でメンバー名を指定する場合には，その名前は同じでないと困る．なお，これらで指定した名前は，出力モジュールの本名である．同じロード・モジュールに別名をつけることも許されている．別名は，最高５つまでつけることができる．何も別名なんかつけなくてもよさそうなものであるが，別名をつけることによって，そのモジュールを実行するさいの“入口点”（最初に実行する命令の位置）を選択できるようになる．名前を変えることによって，入口点が異るようにしておけば，同一モジュールで異った処理法が実現できるわけだ．ともあれ，別名を付けるには，連係編集用制御ステートメントである ALIAS ステートメントを用いればよい．別名も，本名と同じように登録簿に収容される．

連係編集プログラムに診断機能があることは前にふれたことであるが，この診断結果を出力するのに使われるのが**診断用出力データ・セット**（diagnostic output data set）である．普通印刷装置がこれに使われる．診断用出力としては，連係編集プログラムが作り出す“診断用メッセージ”“エラー・メッセージ”と，プログラマーが要求するその他の診断に関する選択事項からなる．プログラマーが要求する選択事項は，EXEC ジョブ制御ステートメントのオペランドでパラメータとして指定する．なお，診断用出力データ・セットのデータ定義名は SYSPRINT であり，やはり DD ステートメントで指定する必要がある．

最後にもう１度第18図を参照していただこう．これまで説明してきた入出力源の機能と相互の関係をこの図をみてまとめていただきたい．それにジョブ制御カードとの関係もとらえてもらおう．ジョブ制御ステートメントのうち，最



第19図　連係編集プログラムによる処理ステップ

初にあるジョブ・ステートメントはジョブの始めを示すのに用いられる. 2番目の EXEC ステートメントでは, 実行するロード・モジュール (プログラム) の名前とさまざまのパラメータが指定できる. 連係編集プログラムの名前 (メンバー名) は LINKEDIT である. 第18図の EXEC ステートメントでは特にパラメータの指定はしていないが, ここで自動呼出し機能の働きを抑止したり, 診断用出力内容の選択指示などをすることができるのである.

4. 連係編集プログラムによる処理ステップ

連係編集プログラムを機能面から考察してみると，５つの処理ステップをとっていることが分る．つまり，初期的処理ステップ，入力処理ステップ，第１次的中間処理ステップ，第２次的中間処理ステップ，最終処理ステップの５つである（**第19図参照**）．

初期的処理ステップは，ジョブ制御プログラムが有していた制御権が，連係編集プログラムに受渡されることによって開始される．このステップにおいて，入力処理ステップ以降のすべてのお膳立てが行なわれるのである．まず，連係編集プログラムが使用するすべてのデータ・セット，つまりジョブ制御ステートメントである DD ステートメントで定義したデータ・セットをオープン（OPEN）するためにデータ管理プログラムの機能を利用する．つまり，すべてのデータ・セットを使用可能な状態にする．

つぎに，プログラマーが EXEC ステートメントのパラメータで指定したオプションや属性を解釈し，**汎用テーブル**（All Purpose Table：略して APT）と呼ばれる記憶域に貯える．ここで，プログラマーが指定することができるパラメータのうち主要なものを簡単にまとめておこう．すでに断片的に説明したようにパラメータとしては，診断出力用のオプション（選択事項），出力モジュ

種　　類	パラメータ	パラメータを指定した場合の働き
診断出力のオプション	LIST	すべての制御ステートメントが，診断出力用データ・セット上にカード・イメージの形式で記録される．
	MAP	出力ロード・モジュールのマップを作り出す．マップには，出力モジュールに含まれるすべての制御セクションの起点．大きさ，入口点などが列記される．
	XREF	出力ロード・モジュールの相互参照表を作る．相互参照表には，モジュールのマップと別の制御セクションを参照するアドレス定数のリストが含まれる．

特殊な処理のオプション	LET	連係編集の過程でエラーが発見されても，出力ロード・モジュールを実行不可能であるとみなさない．
	XCAL	オーバレイ構造のモジュールにおいて，排他的セグメント（同時に主記憶装置に共存しえないセグメント）の間での相互参照が行われていても，出力モジュールを実行不可能とはみなさない．
	NCAL	自動的呼出し機能を働かさない．したがって未解決の外部参照記号があっても．ライブラリーからメンバーの自動呼出しは行われない
スペース割当てのオプション	SIZE=(P1,P2)	連係編集プログラムが使用することのできる主記憶装置の容量と入出力データをいれる作業記憶域の大きさを指定する．
出力モジュールの種々な属性の指定	SCTR と OVLY	SCTR と指定すると，分散ローディングあるいはブロック・ローディングにふさわしい形成のロード・モジュールを出力する． OVLY を指定すると，ロード・モジュールはオーバレイ構造をとることができる．
	RENT と REUS	RENT を指定すると，出力ロード・モジュールには再入可能という属性が与えられ， REUS を指定すると，逐次再使用可能という属性が与えられる．
	NE	出力ロード・モジュールには．外部記号ディクショナリーが出力されない．したがって，このロード・モジュールは連係編集プログラムへの入力モジュールとして使えなくなる．
	OL	この指定によって出力されたロード・モジュールは，LOAD マクロ命令によってしか主記憶装置にローディングできなくなる．

パラメータの指定の仕方（ジョブ制御ステートメントで行う）

例：// EXEC PGM=LINKEDIT, PARM=(XREF, LIST, LET, NCAL)

第20図 主要なパラメータの一覧表とその指定例

ールのさまざまな属性の指定，あるいは自動呼出し機能の働きの抑止などがあ

った．これらパラメータの主要なものをまとめたのが**第20図**である．第20図の下側にはパラメータの具体的指定例を追加しておいた．ここで指定したパラメータにもとづいて，汎用テーブル（APT）を作成するのである．

さらに，初期的処理ステップでは連係編集プログラムで使用するさまざまな記憶域，つまり，各種制御テーブル，作業域，入出力域などに用いるスペースが確保されるのである．

以上のような初期的準備を完了すると，制御権は入力処理ステップのルーチンに渡される．

入力処理ステップは，字義通り連係編集プログラムへのすべての入力データが処理されるステップである．入力データは読取られ，それぞれの妥当性がチェックされたのち，所要の処理がなされる．出力ロード・モジュールの構成要素となるテキストや RLD は，中間的記憶装置（SYSUT 1）に1時的に書き出される．連係編集用の制御ステートメントも処理が行われ，必要なモジュールを呼出しライブラリーや追加の入力源から探し出し，出力ロード・モジュールの1部として組込んでゆく．勿論，テキストや RLD の部分は， SYSUT 1 に書き出される．そのあとで，結合されたロード・モジュールから**合成された外部記号ディクショナリー**（Composite External Symbolic Dictionary：略して CESD）が主記憶装置内に形成される．この CESD は，そのままつぎのステップに引継がれ利用される．なお，**テスト翻訳プログラム**（TESTRAN）と呼ばれるプログラム・テスト用プログラムで利用される記号表（SYM）が入力データとして読まれた場合には，それらをひとまとめにしたうえで，出力ロード・モジュールの1部として直接出力ロード・モジュール・ライブラリーに書き出す．

すべての入力データが受け入れられ，処理が終ると，制御権は第1次中間処理ステップを担当するルーチンへ受けつがれる．この中間処理ステップの基本的機能は，モジュール間のアドレスの調整と中間的出力処理である．アドレスの調整機能とは，出力ロード・モジュールに含まれているすべての外部参照記号に相対的なアドレスを割り振り，モジュールの入口点のアドレスを決定し，必要なら ALIAS ステートメントでプログラマーが指定した別名に対応する入

ロ点のアドレスを割当てる機能である．

一方，出力機能として，出力モジュール・ライブラリーに，このステップで作成したCESDを書き出す．アドレス調整機能で調整した情報の一部をCESDに組み入れたうえで，書き出しを行うわけだ．また，MAPやXREFなどの診断出力用パラメータの指定があると，このステップで処理されて，診断出力用データ・セットに書き出される．

第2次中間処理ステップでは，入力処理ステップで中間的記憶装置に一時的に書き出しておいたテキストとRLDを読み込んできて，テキストの中に含まれているすべてのアドレス定数の再配置を行う．RLDを調べるとテキスト内のアドレス定数が存在する場所がつかめる．複数個のモジュールの組合わせによって，当然，相互のアドレスに相対的変動が生じているので，その影響を受けるアドレス定数の調整が必要なのだ．アドレス定数の再配置を終ると，出力ロード・モジュールを構成するテキストとRLDが出力モジュール・ライブラリーに書き出される．

最終処理ステップでは，出力ロード・モジュールのための登録簿への記入項目を完成する．先述したように出力ロード・モジュールは，区分データ・セットであり，出力モジュール自体は，区分データ・セット内の独立した場所（区分）に収容される．そのモジュールのありかを示したり，モジュールの名前などに関する情報が記録される部分が登録簿（ダイレクトリー）であった．登録簿には，1つの出力モジュールが出力されるごとにそのモジュールの名前やありかを示す記入項目が登録されるのである．

最終処理ステップではこの記入項目を作りあげたうえで，登録簿に記録するのである．出力モジュールがオーバレイ構造をとっている場合には，それぞれのセグメントが出力ライブラリーのどの場所に貯えられているかを示す一覧表を作成して登録簿の一部として書き出す．また，連係編集の過程でエラーが発見され，診断用メッセージが印刷された場合には，その旨の表示が記入項目の中で示される．

以上のような処理ステップを踏んで連係編集が完了するのであるが，1回のランで複数個の出力ロード・モジュールの作成が要請されている場合には，再

び初期的処理ステップに制御権がもどされ，次の入力データの連係編集に入ることとなる．

5. 制御ステートメントとさまざまな機能

これまでの説明では，連係編集プログラムの機能構造をごく概念的に把えてきたのでとかく抽象的にならざるを得なかった．そこで，本節では連係編集プログラムの機能を具体的に把握してもらうために，実体のある例をとり上げて，連係編集プログラム用制御ステートメントと関連性をもたせつつ展開してゆくこととしよう．

1）複数個の目的モジュールを入力する場合

最初に取扱う例は，複数個の目的モジュールを主要入力データ・セットだけから入力し，1つのロード・モジュールに結合する例である．いうまでもなく，主要入力データ・セットからは，連係編集プログラム用制御ステートメントと目的モジュールを一緒に入力しても差しつかえない．もちろん，ジョブ制御カードも同一入力装置から読取らせることも許される．**第21図**に示したのがこの場合である．

```
//              EXEC    PGM=LINKEDIT
                  :
// SYSLMOD      DD      DSNAME=SYS1. USERLIB(INVENT1)
//SYSLIN        DD      *
 ┌──────────────────────────────────────────┐
 │   目的モジュール―1                       │
 ├──────────────────────────────────────────┤
 │   目的モジュール―2                       │
 ├──────────────────────────────────────────┤
 │   目的モジュール―3                       │
 └──────────────────────────────────────────┘
                ENTRY   HAJIME
                ALIAS   ZAIKO1, KOSHIN
                NAME    INVENT1(R)
/*
```

第21図　3つの目的モジュールを1つのロード・モジュールに連係編集

ジョブ制御カードについて言及するのが本章の主旨ではないので，説明上不可避的なものだけの範囲でまず簡単に説明しておこう．EXEC ステートメントのオペランドでは，使用するプログラム名，つまり連係編集プログラムの名前を指定する．さらに，さまざまな連係編集上のパラメータが指定できることは，すでに再三指摘してきた．データ定義名が SYSLMOD である DD ステートメントは，出力モジュール・ライブラリーに関する定義を行うものである．第21図の場合，**データ・セット名**（DSNAME）が SYS1．USERLIB と名づけられているライブラリーを出力モジュール・ライブラリーに適用している．かっこの中に示されている INVENT1 が出力されるロード・モジュールのメンバー名である．データ定義名 SYSLIN が主要入力データ・セットである．オペランドが単に星印（＊）であることは，主要入力データ・セットが，この DD ステートメント以降に置かれていることを意味している．したがって，目的モジュールや連係編集プログラム用制御ステートメントをそのあとに入れておく．第21図の場合には，3つの目的モジュール・デックが置かれているものと仮定している．3つの目的モジュールは別々に言語翻訳して作成されたものであり，これを連係編集プログラムの機能を使って1つのロード・モジュールにしようというわけである．ENTRY，ALIAS，NAME はそれぞれ連係編集プログラム用制御ステートメントである．最後に／＊のステートメントがあるが，これはジョブ制御ステートメントであり，入力データ・セットはこれで終りであることを意味している．

では，連係編集プログラム用制御ステートメントに移ろう．まず，ENTRY ステートメントであるが，これは出力されるロード・モジュールの**入口点**（entry point）を示す．入口点とは，ロード・モジュールが実行にさいして主記憶装置にローディングされたとき最初に制御権が渡される命令の位置である．したがって，オペランドで指定される記号は，プログラム作成にさいして用いられた記号でなければならず，しかも最初に実行すべき命令のラベルとして用いられたものである．注意を要すべきことは，一般の記号名だとコンパイルないしはアセンブルされることによって，相対的アドレスに変換されてしまう．したがって，記号名は，目的モジュールには持込まれないので，ENTRY

ステートメントのオペランドでそのような名を指定してもアドレスと結びつける手がかりはない．そこで，記号名として使えるものには相当な制約があることになる．目的モジュールにも持込まれる記号名はいかなるものであったであろうか．外部記号ディクショナリー（ESD）を記憶している人はすぐに思いつかれることだろう． ESD の中に収容されている記号で，外部から参照できる記号はどのようなものであったか思い出していただきたい．いわゆる外部名と呼ばれるもので，制御セクションに付けられた名前や入口名などであった．このような記号しか ENTRY ステートメントのオペランドでは指定できない．

連係編集プログラムにとって，ENTRY ステートメントは必須のものではない．もし ENTRY ステートメントが省略された場合には，原始プログラムの中で指示した入口点が使われる．つまり，原始プログラムの最後のステートメントである END のオペランドで指定された入口点がそのまま生きるのである． END ステートメントでも入口点を指定していない場合には，出口ロード・モジュールの最初の制御セクションの第1番目の命令のアドレスが入口点とみなされる．

出力ロード・モジュールに別名を与えるのは，ALIAS ステートメントであった．第21図では，INVENT 1 が本名であり，ALIAS で指定したZAIKO 1，KOSHIN が別名である．別名は最高5つまで指定することが可能である．ところで別名として指定する記号が，外部名である場合にはそれは単なる別名だけの機能にとどまらず入口点を変えることを意味する．第21図の連係編集プログラムで出力されたロード・モジュールを INVENT 1 で呼び出して実行するとなるとそのさいの入口点は HAJIME である．ところが，ZAIKO 1 と KOSHIN がいずれも外部名であるとした場合，出力されたロード・モジュールを ZAIKO 1 で呼び出して実行すれば ZAIKO 1 が， KOSHIN で呼び出せば KOSHIN が入口点となるのである．つまり，呼び名を変えることによって，同一ロード・モジュールでも若干異った処理の実現が可能となる．一方別名として単なる記号を使った場合には，その入口点は本名のそれと同じとみなされる．

さて，本名を与える方法であるが，先述したように SYSLMOD の DD ス

テートメントで指定することができる．他の方法としては，NAME ステートメントを使用する．NAME ステートメントのオペランドにその名前を指示する．一般にメンバー名と呼ばれるのが本名である．第21図では INVENT 1 がそれである．SYSLMOD ステートメントでもメンバー名が指定されている場合には同じでないといけない．第21図の NAME ステートメントのオペランドではかつこの中にＲと記されてある．これは，これから出力するロード・モジュールで，すでにライブラリーに登録されている INVENT 1 というモジュールを置換えて欲しいという指示である．これによって，変更・改善されたプログラムで古いプログラムを追放することができるのである．なお，Ｒの指示をしなければ全く新しいロード・モジュールとして登録されることを意味する．

NAME ステートメントには別の機能もある．つまり，１つのロード・モジュールを作り出すさいの入力の区切りを示す役割である．連係編集プログラムには，１つのジョブ・ステップで複数個のロード・モジュールを作る機能があるが，そのような場合，NAME ステートメントにぶっかるとそれまでの目的モジュールと制御ステートメントが１つのロード・モジュールを作成するものであるとみなされて，処理が行われる．したがって，NAME ステートメントの次にさらに別の目的モジュールや制御ステートメントが続いている場合には，それらは次のロード・モジュールを作るのに使われることとなる．このことからお分りのように，NAME ステートメントは必ず１つの出力モジュールを作成するための入力モジュールの最後に置かれる必要がある．しかも，NAME ステートメントだけは，主要入力データ・セットから入力しないことにはその機能は果し得ないこととなる．ともあれ，NAME ステートメントを複数個使うことによって，１つのジョブ・ステップで複数個のロード・モジュールの作成が可能となるのである．

第21図についてまとめよう．これは，３つの目的モジュールを主要入力データ・セットから読取らせて，１つのロード・モジュールに仕立て上げる場合の連係編集のための指定である．連係編集プログラムの処理が終ると，INVENT1 というメンバー名がつけられた１つのロード・モジュールが出力される．ライブラリーに古くからあった INVENT 1 のロード・モジュールはこの新しいモ

ジュールで置換される．INVENT1 というメンバー名で呼び出した場合の入口点は HAJIME である．このメンバーの別名としては，ZAIKO1 と KOSHIN がつけられる．

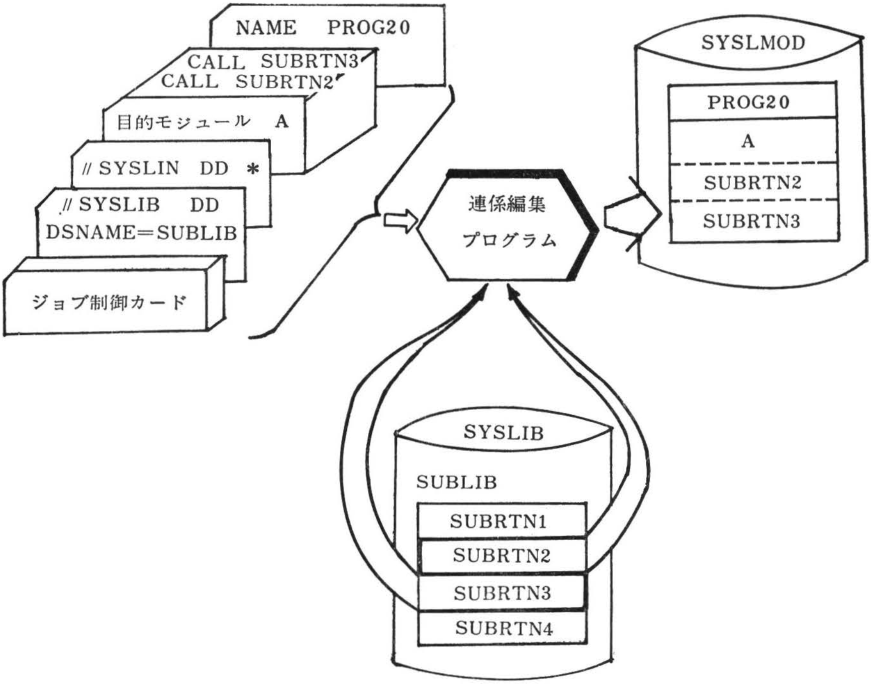
なお，ENTRY と ALIAS ステートメントは目的モジュールの一番前に置いてもよいし，1つの目的モジュールと別の目的モジュールの間に入れても一向に差しつかえない．しかし，NAME ステートメントだけは，入力モジュールの最後に置く必要がある．

2）ライブラリーからの自動呼出し

連係編集プログラムは，主要入力データ・セット以外からも入力モジュールを受け付けることができる．その1つは，自動的ライブラリー呼出し機能の働きを利用して，呼出しライブラリー（SYSLIB）から未解決な記号をもつモジュールを探し出して組み込む方法であった．

自動的ライブラリー呼出し機能を利用して SYSLIB から特定モジュールを組込むには，別に制御ステートメントを用いる必要はない．入力される目的モジュールや制御ステートメントで指定された目的モジュールないしはロード・モジュールの組込みを終ったあと，なおかつ未解決の外部参照記号が残されている場合，自動的ライブラリー機能が働いて呼出しライブラリーから未解決外部参照記号に対応する外部名をもったモジュールが自動的に出力ロード・モジュールに組入れられるからである．ただし，ここでいう外部名は，単に ESD（外部記号ディクショナリー）に登録されていればよいというものではなく，それがモジュールのメンバー名ないしは別名として登録簿に記入されている性質のものでなくてはならない．

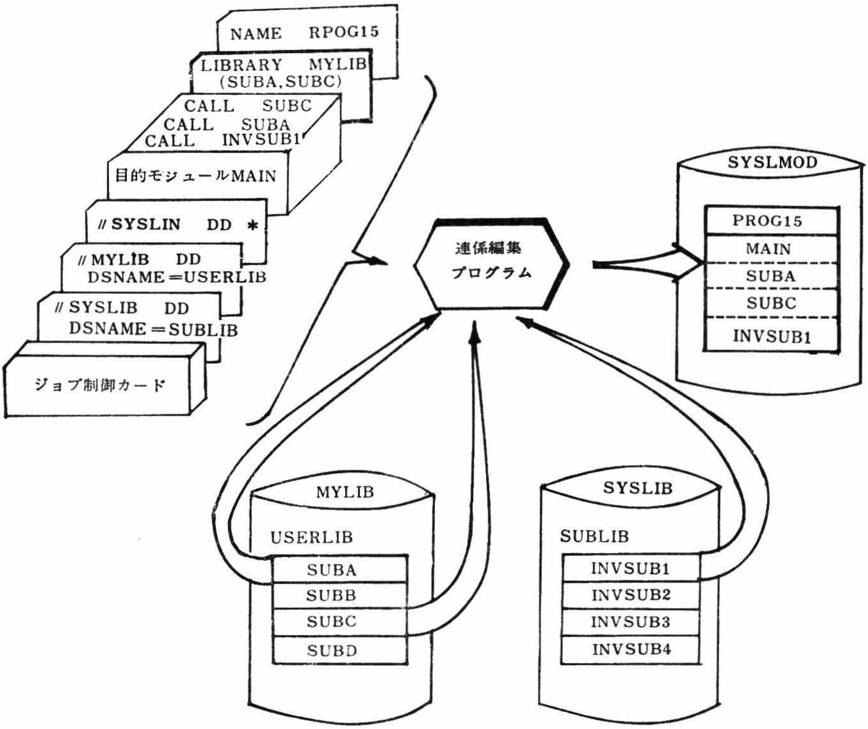
さて，自動的ライブラリー呼出し機能を利用した具体例を**第22図**に示そう．この例でも，ジョブ制御カードの大半は省略して，当面関係するものだけを示している．主要入力データ・セットは，ジョブ制御カードの後に挿入されており，1つの目的モジュールと NAME 制御ステートメントだけから成り立っている．目的モジュールには，2つの CALL マクロ命令が使われており，それぞれ SUBRTN2 と SUBRTN3 と名づけられたサブルーチンを呼んでい



第22図　自動ライブラリー呼出し

る．この入力モジュールが処理されて出力されるロード・モジュールは，PR-OG20 というメンバー名が付けられるであろうことは NAME ステートメントのオペランドで想像がつく．

ところで，目的モジュールAが処理されたあとで，いまだに SUBRTN 2 と SUBRTN 3 という外部参照記号が未解決であると仮定しよう．この場合，自動的呼出し機能が働くこととなる．つまり，連係編集プログラムは，呼出しライブラリー（SYSLIB）の登録簿を探して，SUBRTN 2 および SUBRTN 3 というメンバー名（ないしは別名）をもったモジュールを見つけ出そうとする．該当メンバー名が見つかると，その名前で代表されるモジュールを取り出して，入力モジュール（この場合はモジュールA）と結合することとなる．その結果，**第22図**の右上の出力モジュール・ライブラリー（SYSLMOD）で示したようにメンバー名 PROG20 をもつロード・モジュールは，主要入力データ・セット



第23図　追加ライブラリーからの自動呼出し機能

から読込まれたモジュールAが呼出しライブラリーから自動的呼出し機能によって組込まれたモジュール SUBRTN 2 と SUBRTN 3 の結合されたプログラムとなるのである．

第22図の例は，呼出しライブラリーから未解決記号を満足するサブルーチンを組込む例であったが，**第23図**で示している例は，呼出しライブラリー以外の追加ライブラリーから自動呼出し機能を活用してサブルーチンを組込む例である．呼出しライブラリー以外の追加ライブラリーから自動呼出しをするには，LIBRARY ステートメントを使えばよい． LIBRARY ステートメントのオペランドでは追加ライブラリーのデータ定義名を指定したうえで，その直後のかっこ内に該当ライブラリーから呼び込むモジュール，つまりサブルーチンの名前(メンバー名か別名)を指示すればよい．もちろん，追加ライブラリーは，ジ

ョブ制御用DDステートメントを使ってあらかじめ定義しておく必要がある.

第23図の場合, SUBA と SUBC と名付けられたサブルーチンは追加ライブラリーから呼び込むことになっている. したがって, 主要入力データ・セットからの入力モジュールが処理されたあと, SUBA と SUBC が未解決のままの外部参照記号として残っていると, データ定義名 MYLIB で定義されたデータ・セット, すなわち USERLIB という名前のつけられたデータ・セットの中から SUBA と SUBC というメンバー名をもったモジュールが捜し出され, 出力ロード・モジュールに組込まれることとなる. 追加ライブラリーからの自動呼出しが完了したあと, まだ未解決の外部参照記号が残っていると呼出しライブラリーからの自動的呼出しが行われる. 第23図の例だと INVSUB 1 という外部参照記号はいまだに未解決であるので, 呼出しライブラリーからの自動呼出しが適用される. その結果, INVSUB 1 というメンバー名をもったサブルーチンが出力ロード・モジュールの最後に追加されることとなる.

なお, LIBRARY ステートメントのオペランドにデータ定義名を指定せずにかっこ内に外部参照記号名だけをしるすと, その外部参照記号名は未解決であってもそのまま放置しておいてくれということになる. たとえば, 第23図において LIBRARY 制御ステートメントをつぎのように変更したとする.

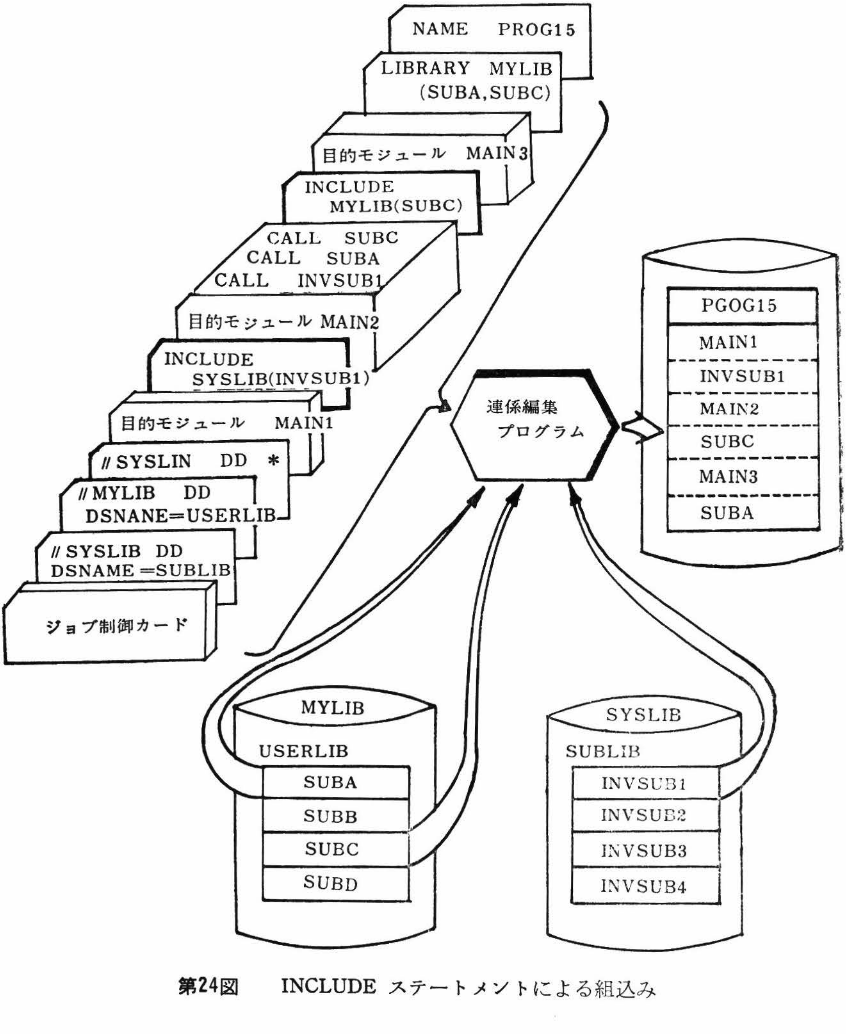
LIBRARY (SUBA, SUBC)

この場合だと, 連係編集プログラムの処理が終ったあとでも, SUBA と SUBC のメンバー名をもったサブルーチンは組込まれないままになる. このような用法は, 当面の連係編集の段階で出力するロード・モジュールには意図的に未解決の外部参照記号を残しておいて, のちの連係編集プログラムの処理のさいに, 所要のサブルーチンを組込みたいといったときに便利である.

3) INCLUDE ステートメントによる組込み

追加される入力源から入力を得るには, LIBRARY ステートメントを使って自動的呼出し機能を活用する方法以外に INCLUDE ステートメントでその場に組込む方法がある.

自動的呼出し機能によって組込まれるモジュールは, 入力データがすべて処



第24図　INCLUDE ステートメントによる組込み

理されたあとで，なお未解決の記号がある場合にだけ先行するロード・モジュールと結合される．したがって，自動呼出しのモジュールは，いわばオープン形式のサブルーチン的機能を果す．これに対して，INCLUDE ステートメントで呼び込まれるモジュールは，その場に，つまり INCLUDE ステートメントがあった場所に組入れられる，クローズド形式である．その例を**第24図**でみよ

う. 第24図では, 主要入力データ・セットから3つの目的モジュールを入力している. それぞれの目的モジュールには, 便宜上 MAIN1, MAIN 2, MAIN 3の名前がつけられている. MAIN 1 と MAIN 2 および MAIN 2 と MAIN 3の間には INCLUDE ステートメントが挿入されている. 最初の INCLUDE ステートメントでは, SYSLIB から INVSUB 1 と名付けられたモジュールを, 2番目の INCLUDE ステートメントでは MYLIB から SUBC と名付けられたモジュールを組込もうとしている. このように INCLUDE ステートメントでは, 組込む対象となるモジュールが区分データ・セット(ライブラリー)に収容されている場合には, オペランドでまずライブラリーに対応するデータ定義名を指定し, かつこの中に組込むモジュールのメンバー名を列記する.

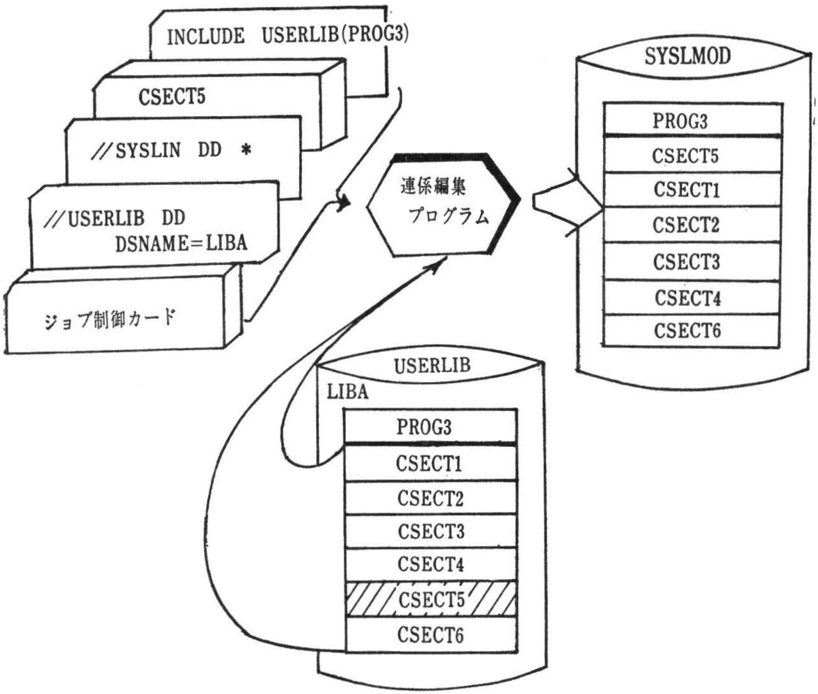
第24図の右上 SYSLMOD に図式化して示した出力ロード・モジュールの順序で推察できるように, INCLUDE で呼び込まれたモジュールはその位置に組み入れられるのである. なお, LIBRARY ステートメントで MYLIB から SUBA と SUBC を自動呼出ししようとしているが, SUBC に関しては INCLUDE ステートメントですでにそのモジュールの組込みが完了しているので, 未解決記号は SUBA だけということになる. その結果, MYLIB から自動呼出しが行われるのは SUBA のモジュールだけである.

INCLUDE ステートメントを使うと区分データ・セットからのモジュールの呼込みだけでなく, 順次データ・セットからの呼込みも可能である. 区分データ・セットの場合には, 目的モジュールでもロード・モジュールでも収容できるが, 順次データ・セットだと目的モジュールだけしか収容できない. 区分データ・セットの場合には, 1つのデータ・セット (ライブラリー) の中に複数個のモジュールを収容しておいて, 必要なモジュールだけをメンバー名で探し出すことができた. しかし, 順次データ・セットの場合にはそのようなことはできない. 順次データ・セットに記録されている全体が入力モジュールとして取扱われざるを得ない. したがって, INCLUDE ステートメントでは, そのオペランドでデータ定義名だけを指定する. その結果, 指定されているデータ・セット内のすべての目的モジュールと制御ステートメントが連係編集プログラムによって処理され, 出力モジュールの1部として組み入れられる.

4）制御セクションの置換編集機能

連係編集プログラムは，プログラムの修正機能をもっている．連係編集プログラムがもっている“モジュールの編集”機能がこれに当る．この機能を用いることによって，容易にプログラムの一部の修正や変更が可能である．修正や変更を要するプログラム全体をコンパイルし直す必要はなく，修正や変更によって影響をこうむる制御セクションだけに手を加えて，元のプログラムの該当制御セクションを置換することができる．また，変更によって別の制御セクションやモジュールから参照される記号，つまり外部記号に支障が生じる場合には特別な制御ステートメントを使って新しい名前に変更したり削除してしまうこともできる．

さて，具体例にもとづいて，制御セクションの置換編集機能を説明してゆくことにしよう．まず，「制御セクションの自動的置換機能」についてふれよう．ここでいう自動的置換機能とは，特別な連係編集用制御ステートメントを使わずして，モジュール内の特定制御セクションを新しい制御セクションと置き換えてしまう機能のことである．制御セクションを自動的に置き換えるには，新しく変更したあるいは修正した制御セクション（または，その制御セクションを含んだ目的モジュール）を置換の対象となる制御セクションよりも前に処理されるような場所に置くように配慮して主要入力データ・セットを構成すればよい．この場合，古い制御セクションの名前と同じ名前を新しい制御セクションにつけておくことが必要である．**第25図**に示したのが，この場合の例である．“LIBA”と名付けられたライブラリーに登録されているモジュールPROG 3は，6個の制御セクションから成り立っている．この6個の制御セクションのうち，5番目の制御セクション CSECT 5 だけを別の修正した制御セクションで置き換えるのが目的である．したがって，主要入力データ・セットには，修正・変更を加えた新しい制御セクションに古い制御セクションと同じ名前「CSECT 5」を付けて入れておく．連係編集プログラムは，同じ名前の付いた制御セクションのうち最初に出てきた制御セクションだけを処理して出力モジュールの一部に組込んでくれる．2番目以降に出てきた同じ名前の制御セクションは無視してしまうので，出力モジュールには含まれない．したがって，第25

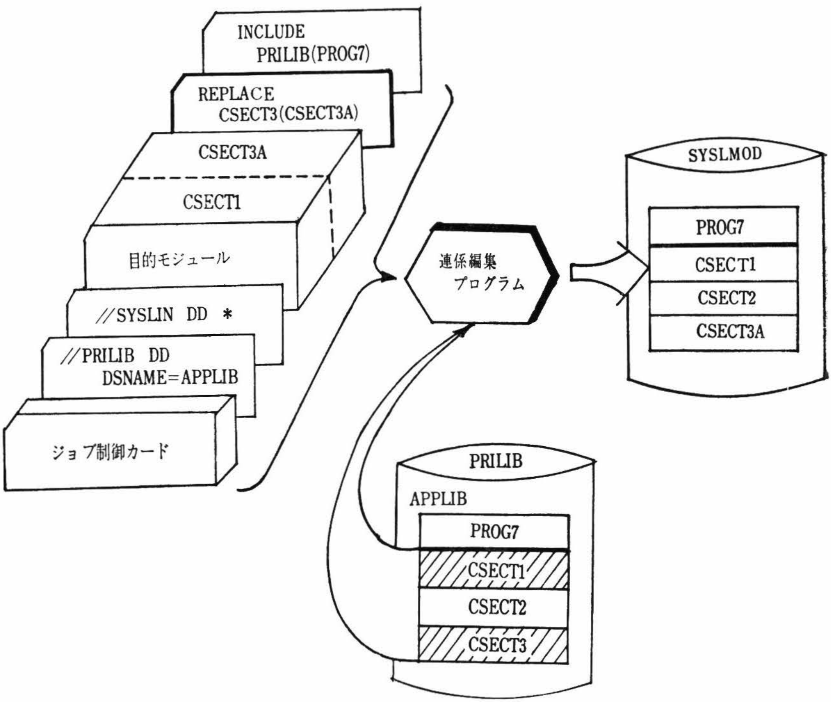


第25図　自動的な置換による編集

図の SYSLMOD に示したように元の CSECT 5 は出力モジュール中には表われないことになってしまう．新しい CSECT 5 は，元の CSECT 5 と同じ位置には組み込まれないが，普通の場合は支障は生じない．制御セクション間はサブルーチン的な連係の考慮がなされている筈だからである．なお，第25図では，INCLUDE ステートメントでライブラリーから“PROG 3”を呼び込んでいるが，自動呼出し機能をもつ LIBRARY ステートメントを使った場合，あるいは CSECT 5 の直後にPROG 5 と同じ構成の目的モジュールを位置づけた場合にも同様の出力モジュールが得られる．

同じ制御セクション名を使う場合には前述のように特定制御セクションの自動的な置き換えができるが，異った制御セクション名をもったもので元の制御セクションを置換するには，連係編集プログラム・ステートメントの1つである REPLACE を用いる．この REPLACE ステートメントを使用した場合の

例が**第26図**である．第26図では，“APPLIB”と名付けられたプログラム・ライブラリーに登録されているプログラム・モジュール PROG 7 を変更修正しようとしている．PROG 7 は，3つの制御セクション CSET 1，CSECT 2，CSECT 3 から成り立っている．このうち CSECT 1 は自動的置換機能で，CSECT 3 は REPLACE ステートメントを使って置換する．主要入力データ・セットからは，元の CSECT 1 を修正するための新しい CSECT 1 と，CSECT 3 と置換すべき CSECT 3 A の名前をもつ2つの制御セクションから構成される目的モジュールが入力される．さらに，CSECT 3 を CSECT 3 A で置き換えるための指示を行う REPLACE ステートメントがつづく． REPLACE ステートメントは，置き換えられるべき制御セクションを含んだモジュールの直前またはそのモジュールを指定する INCLUDE ステートメントの直前に置く必要がある，この例では， INCLUDE ステートメントで，データ定義名 P



第26図　　制御セクションの置換編集

RILIB で指定されうるライブラリーにある PROG 7 というモジュールの組込みを要請している．PROG 7 モジュールにある制御セクション CSECT 3 を置換するのであるから，REPLACE ステートメントはこの INCLUDE ステートメントの直前に置く必要がある．

ともあれ，REPLACE ステートメントのオペランドではまず置換の対象となる元の制御セクション名を指定し，かっこ内で新しい制御セクションの名前を指定する．連係編集プログラムは，まず元の制御セクション（この場合，CSECT 3）をそれが属している入力モジュール（この場合，PROG 7）から削除し，次に新しい制御セクション（この場合，CSECT 3 A）を組入れるようにするのである．

第26図の場合，CSECT 1 は自動的置換機能で新しい制御セクションに置きかわり，CSECT 3 は REPLACE ステートメントの機能で置換されて，右側の SYSLMOD で示したようなロード・モジュールが完成するのである．

REPLACE ステートメントには，置換の機能以外に削除だけの機能もある．置換の機能は，さきにふれたように，まずこのステートメントのオペランドで指定した元の制御セクションを削除したあとで，かつこの中で指定した制御セクションを組込むことによって置換を達成している．したがって，この置換機能の前の部分，つまり，削除の機能だけを使えばよい．そのためのステートメントの指定では，オペランドのかっこ内の指定を省略すればよい．例えば第26図の例で REPLACE ステートメントを「REPLACE CSECT 3」とすれば削除の機能だけが働く．したがって，出力モジュールには，新しく置きかわった CSECT 1 と元の CSECT 2 だけが出力される結果となる．この方法によって，不要となった制御セクションを取り除くことができるようになる．

このようにして，特定制御セクションを新しい制御セクションと置きかえたり，削除したりすることができるのであるが，新しい制御セクション内で使われている外部記号名と元のモジュールで使われているそれとが異ってしまうため不都合が生じる恐れがある．もちろん，修正・変更のための新しい制御セクションを作成するさいこの点に十分留意した外部記号を使用すれば問題ないのであるが，作成後になってそのくい違いに気づく場合も少なくなかろう．その

ような場合には，CHANGE ステートメントを使って，制御セクション名，入口点あるいは外部参照名などの変更が可能である．たとえば，「CHANGE CSECT 3 (CSECT 3 A)」という指定をすれば，元の外部参照名 CSECT 3 は新しい外部参照名 CSECT 3 A に変更できるのである．この CHANGE ステートメントは， REPLACE ステートメントと同じように変更する制御セクション名，入口点あるいは外部参照名を含んでいるモジュールの直前ないしはそのようなモジュールを指定する INCLUDE ステートメントの直前に置く必要がある．

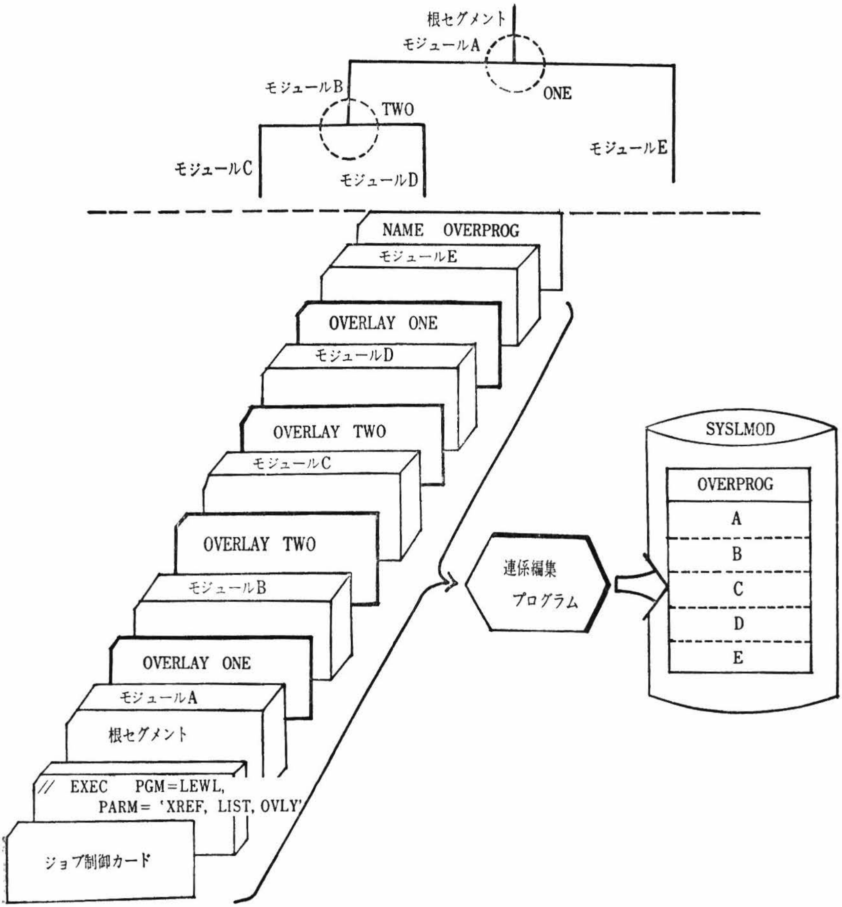
5）オーバレイ構造のプログラムを編成する機能

オーバレイ構造のプログラムを作るにあたって，そのプログラム作成者は，つぎのことを行う必要がある．まず，しっかりとした“樹木式構造”をプランすることである．その場合，このプログラムで使用可能な主記憶装置の容量，各セグメントの使用頻度，相互の依存関係と従属関係あるいは1つのパスの中での制御権の移動の仕方と別のパスや区域への制御権の移動の仕方などを考慮したうえで，樹木式構造を組立てることである．一応構造が決定したとなると各セグメントのプログラム作成に着手することになるが，あるセグメントから別のセグメントへの制御権の受渡しに関する命令を組み入れておく必要がある．この目的に使用される命令には，種々のものがあるが，最も代表的なものは CALL 命令であろう．この命令が実行されると，指定された記号をもつセグメントがあらかじめ主記憶装置にローディングされ，そのあとでそのセグメントの所要の個所に制御権が渡る．この命令以外にも，あるセグメントがローディングされている最中に別のセグメントでの処理を併行的に行って，効率をより高めてゆくための命令などがあるが，本節の主旨からそれてしまうので特に説明はしない．ともあれ，プログラム作成段階でセグメント相互間での制御権やデータの受け渡しに関する考慮がなされている必要がある．

以上のことについて検討がなされたうえでプログラムが作成されていると，あとは樹木式構造を参考にしながら，各セグメントと各区域が樹木式構造の中で占める相対的位置や各セグメント内での制御セクションの相対的位置を連係

編集用ステートメントを作って指定してやれば，意図通りのオーバレイ構造が完成するのである．

このことを具体例にもとづきながら説明してゆくことにする．その第1は，連係編集プログラム用制御ステートメントと入力モジュールのすべてを主要入力データ・セットから入力する場合の例である．この場合を図式化して示したのが**第27図**である．オーバレイの樹木式構造は上側に示した通りである．各々のセグメントが樹木式構造の中に占める相対的位置を連係編集プログラムに知らせる必要があるが，各セグメントには名前が付けられていないので，そのセグメントの起点を示す記号名を与える必要がある．そのために結節点（ノード・ポイント）に記号名を付けることにする．ONE と TWO がそれに当る．この名前を連係編集プログラム用制御ステートメント OVERLAY のオペランドで指定することによって各セグメントの相対的位置を知らせるのである．したがって，OVERLAY ステートメントのオペランドで指定する記号はそれぞれのセグメントの起点を意味することになる．第27図のオーバレイ樹木式構造では，根セグメントがモジュールAであり，結節点 ONE を起点としているセグメントは，モジュールBとEである．さらに，結節点 TWO を起点としているのは，モジュールCとモジュールDである．この樹木式構造を念頭にして作り上げた主要入力データ・セットが第27図の左側の一連のカードである．いうまでもなく，主要入力データ・セットに先行して置かれているのがジョブ制御カードである．ジョブ制御カードの中の EXEC ステートメントの第1オペランドでは，このジョブ・ステップで実行するプログラム名，つまり連係編集プログラムの名前を指定する必要があることは，これまで再三指摘した通りである．2番目のオペランド，つまり PARM=の個所では，出力ロード・モジュールの属性が指定できる．これから出力しようとしているロード・モジュールはオーバレイ構造をとるのであるから，パラメータとして OVLY を指示する必要がある．OVLY という属性を指定すると，ロード・モジュールは，オーバレイ構造に適した**ブロック・ローディング**（block loading）の形式をとってくれる．ブロック・ローディングの形式とは，出力モジュールを主記憶装置内の連続した記憶域に記憶できるような形式である．オーバレイ構造をと



第27図　オーバレイの例（主要入力だけ）

る場合には，主記憶装置内の互に連続していないいくつかの記憶域に分散してローディングする方法は，その性格からいって無理なことである．したがって必ずブロック・ローディングの属性が要求されるのである．ただし，複数個の**区域**（region）に分割した形式の樹木式構造にした場合には，各々の区域は必ずしも連続した記憶域である必要はない．区域内のセグメントが記憶される記憶域だけはどんな場合でも連続している必要があるが，この属性を要求するの

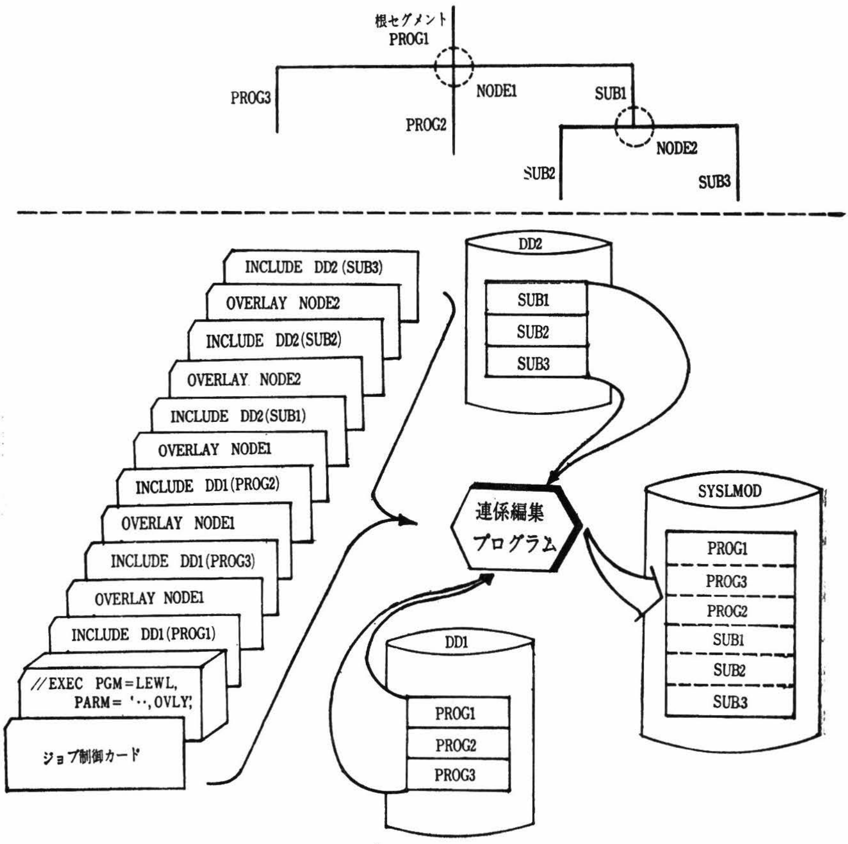
が OVLY パラメータである．

さて，主要入力データ・セット内の入力モジュールと OVERLAY ステートメントの入力順序をながめてみよう．制御ステートメントおよび入力モジュールからなる入力データ・セットの一連の順序は，オーバレイの樹木式構造を構成するセグメントの順序，すなわち，樹木式構造の上から下へ，そして左から右への順序でなければならない．

それゆえ，第27図において，まず第1に根セグセントを構成するモジュールAが，つぎにモジュールBの起点を示す“ONE”をオペランドとした OVERLAY ステートメントがつづき，そのセグメントに当るモジュールBをしたがえる．モジュールBのセグメントは，結節点“TWO”を起点として2つのセグメントに分かれる．したがって，まず，モジュールCのセグメントを規定するために OVERLAY のオペランドを“TWO”としたステートメントを，つづいてモジュールCを置き，再び“TWO”を起点としてモジュールDを置く．ここまでくると，残りは，モジュールEを持つセグメントだけである．ところが，このセグメントは，結節点“ONE”としているのであるから，OVERLAY ステートメントでこのことを示したうえでモジュールEをつづければよいのである．入力データ・セットを作成するさいにしては，何時でも「樹木式構造のセグメントの順序，つまり上から下へ，そして左から右へ」を忠実に守って指定してゆけばよい．

その結果，第27図に示したごとく，OVERPROG と名付けられたオーバレイ構造のロード・モジュールが出来上るのである．SYSLMOD には，入力モジュールが出力モジュールにどのような順序で表われるかしか示していないが，実際にはオーバレイにとって必要となる種々の情報を集約した制御テーブルを各セグメント単位で出力してくれるので，このロード・モジュールが実行時にオーバレイを行うにあたって全く支障が生じることはなくなる．

第2の例は，オーバレイ構造を構成するすべてのモジュールが，あらかじめライブラリーに登録されている場合である．**第28図**がその全体の概略である．樹木式構造は例によって上側に示してある．根セグメントを構成するモジュールは PROG1 である．根セグメントに接続するセグメントは，3つに分岐す



第28図　オーバレイの例（ライブラリーからの入力）

る．“NODE 1”を起点に PROG 3 のモジュールから成るセグメント，PROG 2 のモジュールから成るセグメントおよび SUB 1 から成るセグメントの3種類である．さらに，SUB 1 にひきつづいて，2つのセグメントがある．この2つのセグメントは“NODE 2”を起点としており，SUB 2 のモジュールから成るセグメントと SUB 3 とから成るセグメントである．

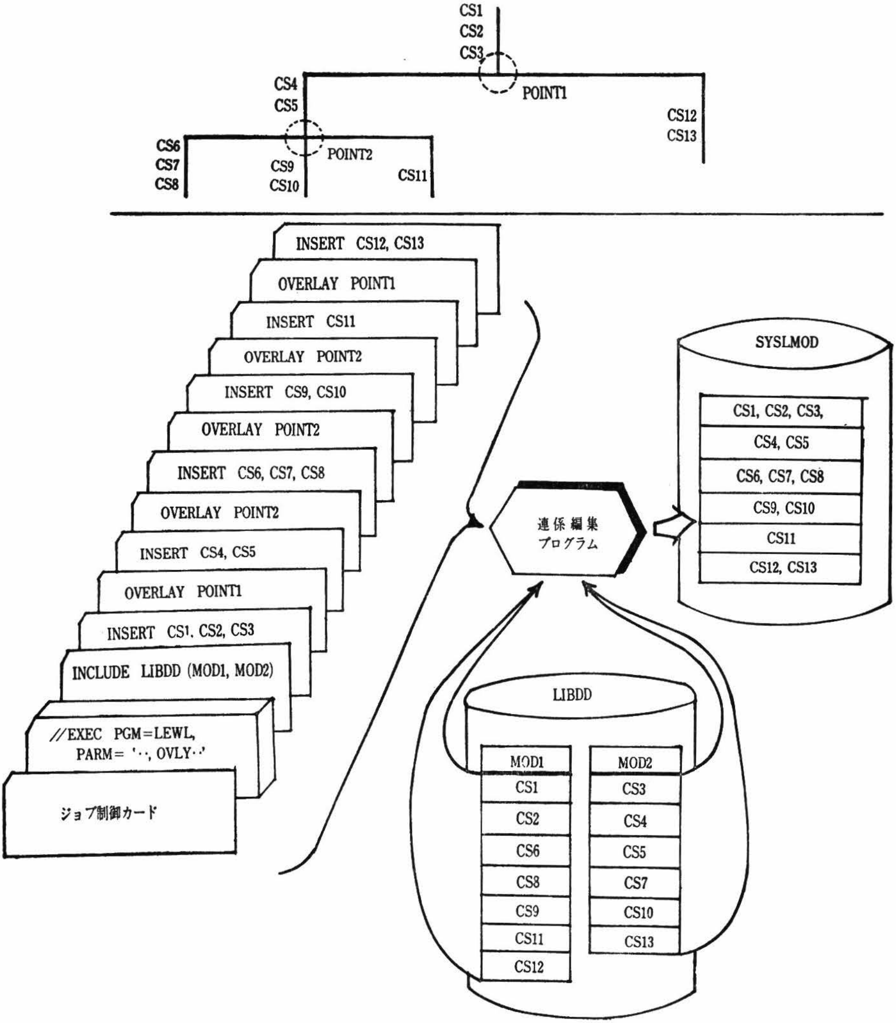
この樹木式構造のオーバレイを達成するための連係編集プログラムの機能と制御ステートメントの指定を図示したのが下側である．図に示したように入力モジュールは，すべてライブラリーにあらかじめ収容されている．データ定義

名 DD 1 と DD 2 のライブラリーである．これら2つのライブラリーの定義はジョブ制御ステートメントで行っておく必要があることはもちろんであるが，特に図の中では示さなかった．主要入力データ・セットは，ジョブ制御ステートメントと連係編集プログラム用ステートメントだけから出来上っている．ジョブ制御ステートメント EXEC のオペランドで OVLY の属性を指示しておく必要のあることはまえにふれた通りである．

連係編集プログラム用制御ステートメントは，INCLUDE ステートメントと OVERLAY ステートメントの組合わせだけから成り立っている．各々の INCLUDE ステートメントは，それぞれのゼグメントを構成するモジュールをライブラリーから呼び込んでいるにすぎない．あとは，OVERLAY ステートメントを樹木式構造にしたがって，上から下へ，左から右への順序で，それぞれのセグメントの相対的位置を指示しているのである．上側の樹木式構造と対比しながら，制御ステートメントを追跡してゆけば，その意味は納得していただけよう．

1つのモジュールを構成しているいくつかの制御セクションを，異ったセグメントに分散して組込んでいくためには，これまで説明した2つの例では解決しきれない．それに，最初の OVERLAY ステートメントより前に出てくる制御セクションおよび自動呼び出しの機能で取り出されてきたすべての制御セクションは，根セグメントに組入れられてしまう．これらの制御セクションを別のセグメントに位置づけし直すのに使われるのが， INSERT ステートメントである． INSERT ステートメントを用いることによって，オーバレイ構造をとるプログラムの中の適切なセグメントに制御セクションを組込むことができるようになる．その具体的用法を示したのが**第29図**である．この例では，データ定義名 LIBDD のライブラリーに登録されているモジュール MOD 1 と MOD 2 を使って，上側に示したようなオーバレイ構造をとろうというわけである．ここに示した樹木式構造では，それぞれのセグメントがどの制御セクションから構成されるかを記号名で表わしている．

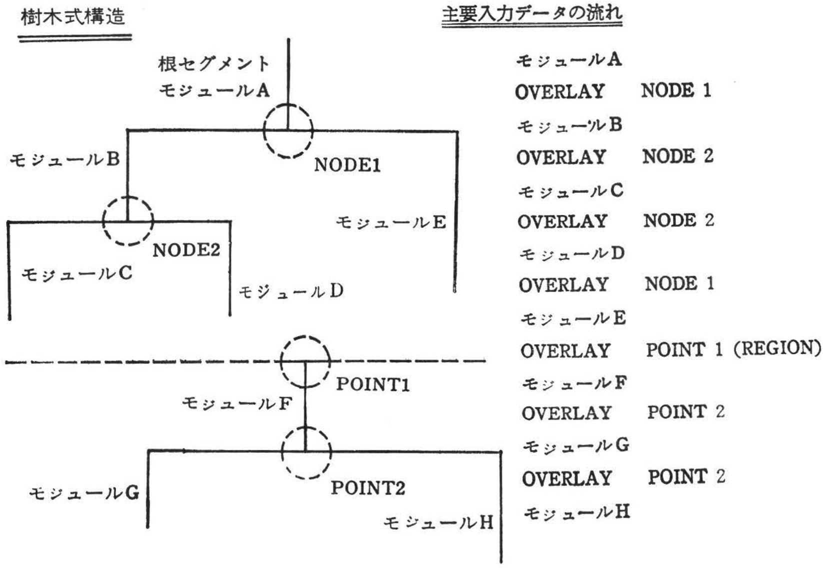
図の左側に示した一連のカードが，ジョブ制御カードと主要入力データ・セットである．まず INCLUDE ステートメントを使って，MOD 1 と MOD 2



第29図　　オーバレイの例（INSERTステートメントを使用したとき）

を呼び込み，そのあとで INSERT ステートメントを使って，各セグメントに適切な制御セクションを位置づけているのである．INSERTステートメントのオペランドでは，その個所に位置づける制御セクションの名前をカンマで区切って順番に指定しておけばよい．この例でおわかりのように INSERT ステートメントは，オペランドで指示した制御セクションが組入れられるセグメントの起点を指定する OVERLAY ステートメントの次に置く必要がある．また該当制御セクションを根セグメントに位置づけるのには，最初の OVERLAY ステートメントの前に INSERT ステートメントを置くことになる．この例では，INCLUDE ステートメントを主要入力データ・セットの先頭に置いているが，逆に最後に置いても，INSERT ステートメントの働きで同じ結果が得られる．なお，INCLUDE ステートメントの代りにその場所に目的モジュールが置かれていても結果は同じである．

これまでみてきた３つの例はいづれも１つの区域からなるオーバレイ構造であったが，最後に複数個の区域を利用する場合のオーバレイ構造について説明



第30図　オーバレイの例（２つの区域をもつ場合）

しておこう．第30図の左側に示した樹木式構造は，２つの区域を利用する構造である．この場合，上側の樹木式構造から得られる“パス”と下側の構造から得られる“パス”は独立した記憶域（つまり区域）にローディングされる．

“パス”とは，以前にも説明したことであるが，１つの区域の中で対象となるセグメントと根セグメントを結ぶ複数個のセグメントのことである．第30図において，モジュールＤを取り上げた場合，モジュールＤ，モジュールＢ，モジュールＡの各セグメントでパスが構成されることになる．モジュールＣを取り上げると，モジュールＣ，モジュールＢ，モジュールＡの各セグメントで，モジュールＥの場合だと，モジュールＥとモジュールＡでパスが構成される．これら３組のパスは，１つの記憶域をオーバレイによって利用する．他方，モジュールＧとＦのセグメントからなるパス，モジュールＨとＦからなるパスは，別個の独立した区域を共用するわけである．

２つの区域は，連続した記憶域である必要がないことから，記憶域の有効利用が計れる．それに一方の区域から，それ自身のパスには存在しないセグメントを指すこともできるし，次に実行すべきパスを他方の区域にローディングしている最中に，併行して別の区域での処理が続行できることから，全体としての処理効率も高まる．

このような理由から，あえて複数個の区域を利用するような樹木式構造をとった方が有利な場合がしばしばある．

話をもとにもどそう．第30図のような２つの区域を利用するオーバレイ構造を指定することは極めて簡単なことである．これまで説明してきた手順と全く同じ方法で上側の構造について指示する．新しい区域についての定義を開始するにあたっては，その区域の最初のセグメントの起点を定義する OVERLAY ステートメントのオペランドのかっこ内で REGION という単語を指定することで足りる．このステートメントで新しい区域の起点を連係編集プログラムに知らせるのである．第30図の右側に示してあるのが，この場合の形式である．

３つ以上の区域を用いる場合も全く同じ考えである．新しい区域の最初のセグメントの起点を指示するときだけ，OVERLAY ステートメントのオペランドに REGION という単語をかっこで付け足せばよい．あとは，「上から下へ，

左から右へ」の規則で各セグメントを定義してゆけばよいのである．

以上，連係編集プログラムのさまざまな機能構造を制御ステートメントの指定の方法を通して概観してきた．

ここで例に取り上げたと全く同一の形式での指定が行われるとは限らないがどんな連係編集プログラムでもおおよそ以上のような機能を何んらかの指定方法によって達成するように設計されているはずである．

6. ローダーの機能

連係編集プログラムは，目的モジュールあるいはロード・モジュールを入力して，実行可能なロード・モジュールを作成してくれた．出力されたロード・モジュールは，直接アクセス記憶装置上のプログラム・ライブラリーに登録される．この連係編集の過程で，連係編集プログラムのさまざまな機能を利用するために制御ステートメントを使ってその要求を行うのである．しかし，連係編集プログラムを使用するとなると，いくつかの不合理さがある．まず第1は連係編集のためにわざわざ余計に1つのジョブ・ステップを必要とする．もし目的モジュールの状態でプログラムの実行が可能であるとしたら，この連係編集プログラムのステップは全く余計のものと言えよう．第2世代のコンピュータやオペレーティング・システムを使わない第3世代の小型コンピュータなどでは，言語翻訳を行ったプログラムは即実行可能なプログラムとして通用している．連係編集プログラムなどと呼ばれるものは全く存在しないのである．その代り，プログラムを分割して作成するとか，出来合いのサブプログラムを自由に結合して用いるとかあるいは複雑なオーバレイ構造をもったプログラムを作り上げる場合の柔軟性はほとんどないし，言語翻訳の段階で固定的に主記憶装置のアドレスが割り当てられてしまうので，ダイナミックなローディングは望めない．とはいうものの，余計な処理ステップを要しないのであるから，全体としての処理効率はそれだけ高くなるといえるのである．

第2に，連係編集プログラムを使うには，それに必要な制御ステートメントを作り上げる仕事が入る．種々の機能を活用するためには，これは当然なことではあるが，それほどの機能を必要としないときでさえある程度の制御ステー

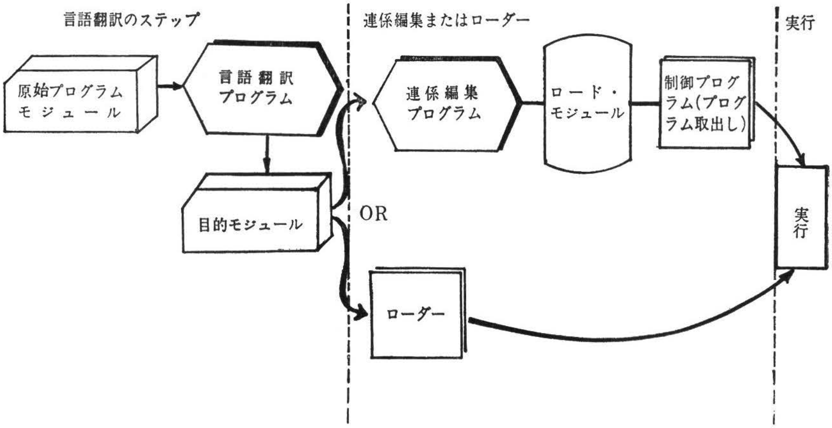
トメントを作らざるを得ないとしたら，やはりわずらわしいことである．それに連係編集プログラムを使うと，中間処理結果を一時的に保存しておくバッファ・データ・セットであるとか，あるいは出力ロード・モジュールを実行時まで貯えておくライブラリーも必要となる．それだけ余分な補助記憶装置を用意しておかなければならないとは，不合理なことである．

このような理由から，連係編集プログラムに代るもの，つまり，余計なジョブ・ステップを必要とせず，わざわざ制御ステートメントを作成する手間のかからない，そして，補助記憶装置のスペースを節約することのできるようなプログラムが指向されるのは当然である．その結果として，設計されたのが**ローダー**（loader）と呼ばれるプログラムである．ローダーもオペレーティング・システムを構成する一要素であり，処理プログラムの1つである．ここでいうローダーは，単に実行可能なプログラムを主記憶装置にローディングする機能をもった一般的なローダーではない．これは，制御プログラムの一部をなすローダーであって，処理プログラムではない．ここで取り上げるローダーは，入力された目的モジュールやロード・モジュールを処理し，1つの完成したロード・モジュールを作り上げ，かつ，そのロード・モジュールを同一ジョブ・ステップ内で実行させることのできる機能をもった処理プログラムである．連係編集プログラムの基本的機能と，同一ジョブ・ステップ内で高い効率のプログラム・ローディングを達成し得る機能を兼ね備えるように設計されているのがローダーである．**第31図**で示したようにローダーを利用した場合には入力モジュールを主記憶装置内で処理し，かつ即実行可能な形式で主記憶装置に出力ロード・モジュールを配置してくれるので，時を置かずその場で実行ができるのである．つまり，ローダーが主記憶装置の一角を占めていて，入力モジュールを次々と処理して，その処理した出力モジュールを主記憶装置の別の一角にどんどん並べていってくれるわけである．あたかも，連係編集プログラムの機能とプログラム・ローディングの機能を同時併行的に達成してくれているような状況を作り出してくれる．連係編集プログラムは，出力ロード・モジュールをライブラリーに登録してくれるが，ローダーにはそのような機能はない．全くの一発主義であり，出力ロード・モジュールの実行が終ってしまうとそのロー

ド・モジュールは消されてしまう運命にある．したがって，くり返し使われるようなプログラムを作成するにあたって，ローダーを使うことは得策ではない．プログラム・テスト段階におけるプログラムの実行であるとか科学技術計算のように一発主義のプログラムで，後になって再度利用することのないようなプログラムの実行にさいして有効であるといえる．ただし，主記憶装置にはローダーと出力ロード・モジュールが一時的に同居することになるのでローダーが占める記憶域だけ余分なものが必要となる．

ローダーには，連係編集プログラムの基本的機能とプログラム・ローディングの機能があるが，これについてもう少し詳しくふれておこう（**第32図参照**）．

左上に示したように，入力モジュール相互間での外部参照記号を解決し，1つのロード・モジュールにしてくれる．この場合の入力モジュールは必ず主要入力データ・セットに存在することを条件とする．連係編集プログラムのように他のライブラリーから INCLUDE ステートメントで呼び込んできて結合することはできない．なぜなら，ローダーの使用にさいしては，連係編集プログラム用制御ステートメントのようなものは使わないことが設計目標となっているからである．なお，主要入力データの形式は，目的モジュールであってもよいし，連係編集プログラムで出力されたロード・モジュールの形式であっても



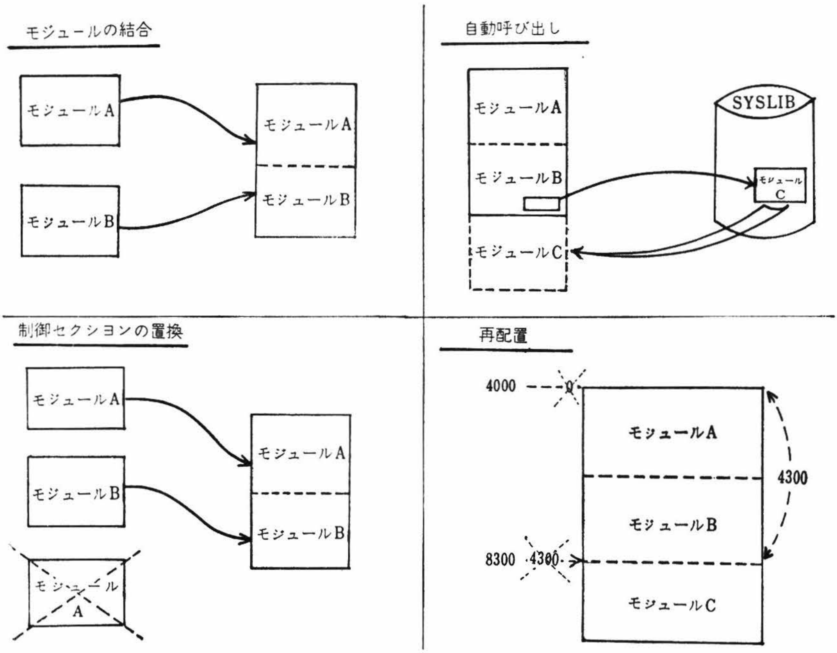
第31図　連係編集プログラムとローダー

構わない．

主要入力モジュールを処理したあと，未解決の外部参照記号が残っている場合には，自動ライブラリー呼出し機能を働かせることができる．この機能を利用することで，システム・ライブラリー（SYSLIB）からサブルーチンやサブプログラムを組入れることが可能となる．さらに，制御セクションの置き換えも可能である．この置換方法は，連係編集プログラムにおける制御セクションの自動的置換えの方法と全く同じである．つまり，同じ名前の制御セクション（またはモジュール）が入力された場合，最初のセクションを出力モジュールに組入れ，２番目以降は削除してしまう方法である（**第32図左下**）．

さらに，ローダーは，出力ロード・モジュールに含まれているアドレス定数に実番地を割当てることによって再配置を達成すると同時に，そのロード・モジュールを実行するように制御権を受渡してくれる．

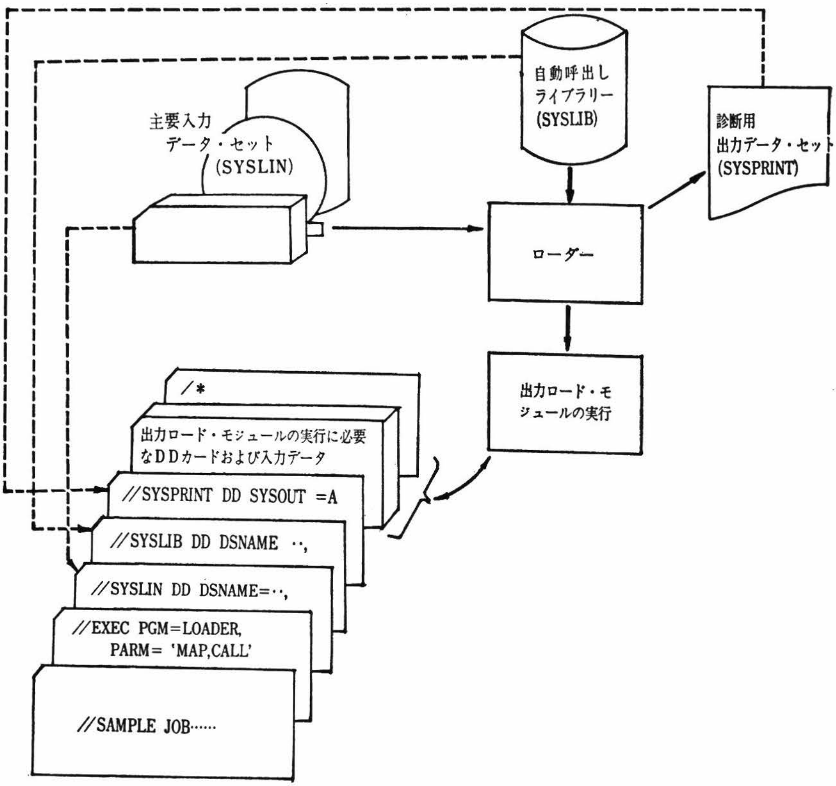
このように，ローダーにはプログラム・ローディングの機能と基本的な連係



第32図　　ローダーの機能

編集プログラムの機能は含まれているが，オーバレイ構造を作るとか，出力ロード・モジュールに名前をつけるとか，SYSLIB以外のライブラリーから INCLUDE ステートメントや LIBRARY ステートメントを駆使してさまざまなモジュールを組入れるといった機能はない．しかし，連係編集プログラムが持つ基本的機能とロード・モジュールの実行を同一ジョブ・ステップ内で達成できることから余分のシステム・オーバヘット時間を排除できること，および制御ステートメントを処理したり，オーバレイ構造のような複雑な処理を回避できることで，その処理効率は一段と高まるのである．それに主記憶装置内だけで出力ロード・モジュールが構成できることから，余計の補助記憶装置を必要としない．

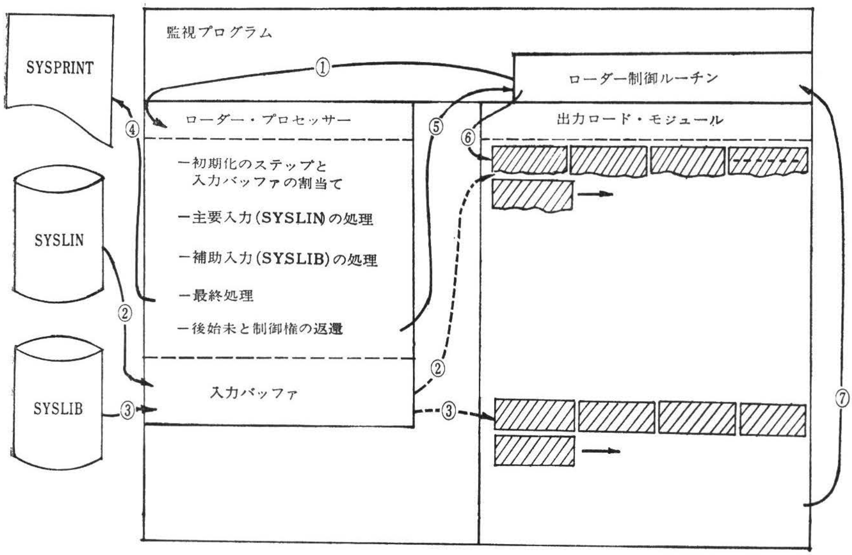
第33図に図示したように，中間的記憶装置であるバッファア・データ・セットも不要であるし，出力モジュール・ライブラリーも使わない．ローダーが必要なのは，主要入力データ・セット（SYSLIN）だけである．ただし，自動呼び出しの機能を利用する場合には，自動呼び出しライブラリーが，ローダーの処理過程において診断用メッセージを必要とする場合には，診断用データ・セットが必要となることは言うまでもない．ローダーを使って出力ロード・モジュールを作り上げ，その出力ロード・モジュールを実行するには，第33図の左下方に示したようなジョブ制御カードの指定が必要である． EXEC ジョブ制御ステートメントで，“LOADER”と指定することが必要だ．PARM では，連係編集プログラムの指定の場合と同じようにいくつかの選択事項(オプション)の指示ができる．“MAP” と指定すると，出力ロード・モジュールのマップ，つまり，そのモジュール内の外部参照名と，それぞれに割り当てられた主記憶装置の実番地がリストされる．これは，診断用出力データ・セット(SYSPRINT) 上に印刷される． “PRINT” という指定をすれば診断用メッセージもこのデータ・セット上にはき出される． “CALL” の指定が，自動呼び出し機能を使うことを要求しているのである．この指定をすることは SYSLIB を利用することであるから，後続の DD ステートメントでこのライブラリーのの定義が必要である． SYSLIN の定義は，ローダーを使う限り必ず必要である．SYSPRINT の定義は，先述の “MAP” や “PRINT” を EXEC ステートメントの



第33図　ローダーへの入出力源とジョブ制御カードの指定例

パラメーターで指定したときは必要となる．ローダーに関するジョブ制御ステートメントはこれだけでこと足りるのである．あとは，ローダーによって出力され実行されるロード・モジュールに必要となる入出力データ・セットを DD ステートメントで定義すればよい．それに，入力ジョブ・ストリームから入力データ・セットを読込んで処理するとしたら，そのデータをDDステートメントの後に続ければよいのである．連係編集プログラムを利用する場合のような制御ステートメント（NAME，ENTRY，INCLUDE ステートメントなど）のようなものは，ローダーにとっては全く不必要である．

さて，最後にローダーの処理過程を，コントロールの流れにそって説明して



第34図　ローダーによる制御権の論理的流れ

おこう．ローダーは，大別して，制御ルーチンと処理ルーチンに分かれる．ローダーの制御ルーチンとは，オペレーティング・システムの監視プログラムとローダーの処理ルーチンないしは出力ロード・モジュール間のインターフェイスとして作用する．ローダーの処理ルーチン，つまり，ローダー・プロセッサーは，ローダー制御ルーチンからパラメータと制御権を受けとって，入力モジュールの処理をし，出力モジュールを作り上げてくれる．この様子を示したのが，**第34図**である．

ローダー制御ルーチンは，まず，ローダー・プロセッサーの部分を主記憶装置にローディングし，ジョブ・スケジューラーから引渡された制御情報（パラメーター）と制御権をローダー・プロセッサーに受渡す．ローダー・プロセッサーは，引き続く処理ステップの準備のための初期化のステップをとる．この初期化のステップには，ジョブ・スケジューラーから引渡された制御情報の分析，主記憶装置の初期化（つまり，入力バッファの設定や出力ロード・モジュールの収容個所の設定)，ローダーが使う入出力データ・セットなどのオープンをする．

そのあとで，主要入力データ・セット（SYSLIN）から入力モジュールを呼び込んで処理を行っていく．ちょうど連係編集プログラムが主要入力データ・セットを処理してゆくステップと同じであるが，唯一の違いは，連係編集プログラムの場合には，その処理結果をいったんバッファ・データ・セットに収容するのであるが，ローダーの場合には，処理結果を主記憶装置内の出力ロード・モジュール域に順番に配列してゆくことである．

この主要入力データの処理が終ったあと，未解決の外部参照記号が残っていると，自動呼出し機能が働く．未解決の記号に等しい記号をもったモジュールを自動呼出しライブラリー（SYSLIB）から探し出し，入力バッファで処理したあと，出力ロード・モジュール域に記憶してゆくのである．すべての未解決外部参照記号の処理が終ると，最後の処理ステップに入る．このステップで，診断用出力データ・セットにメッセージを書き出す．そのあとで，制御権を再びローダー制御ルーチンに返却する．

ローダー制御ルーチンは，ローダー・プロセッサーが占有していた主記憶装置を他にあけ渡したあとで，完成したロード・モジュールを実行するために，制御権をそのモジュールに渡す．この結果，ロード・モジュールの実行が開始される．ロード・モジュールの実行が終了すると，制御権は，再度ローダー制御ルーチンに戻され，このジョブ・ステップが完了することとなる．

これが，ローダーの処理の流れである．このような方法で，連係編集プログラムの不得手な領域をカバーするために出現したのがローダーである．

オペレーティング・システムを構成するどのプログラムを取り上げてみても万能なものはない．どこかに限界あるいは不合理な部分がある．この限界あるいは不合理な部分を克服するため，そのギャップをうずめるために新しい機能が追加されたり，新しいプログラムを組み込むことによって絶えず成長・拡大してゆくのがオペレーティング・システムの姿である．

第 5 章

データ管理の諸機能とその構造

一. 制御プログラムの三大管理機能

情報処理システムの主要な目標は，いうまでもなく，目的に合わせた“データの処理を行う”ことにある．この目標を経済的かつ能率的に達成するには，システムを形成しているハードウェア諸機能のすべてを可能なかぎり“こき使う”ようにすることである．コンピューティング・システムの生産性を向上させるという目標のために出現したのがオペレーティング・システムであることはこれまでに指摘してきた通りである．

オペレーティング・システムはCPU，主記憶装置，入出力装置あるいはシステムの一部をなしているプログラムやデータなどの利用可能なすべての諸資源を巧妙に管理することによって，コンピューティング・システムの生産性の向上に貢献する．ここでいう諸資源の管理とは，コンピューティング・システムに投入された仕事（ジョブ）に対して無駄の1番少ないような方法で，その仕事の遂行に必要なさまざまな資源を割り振ってゆくことである．コンピューティング・システムで処理すべき仕事は非常に多様でかつ多量なものであるから，同時にすべてを遂行することは，システム資源が制約されていることから，とうてい不可能である．そこで，仕事はいくつかの独立したジョブに分割したうえで，コンピューティング・システムに投入される．投入されたジョブをオペレーティング・システムが受取ると，そのジョブに必要な資源，つまり主記憶装置や入出力装置などを割り当てる．投入されたジョブに必要な諸資源が割り

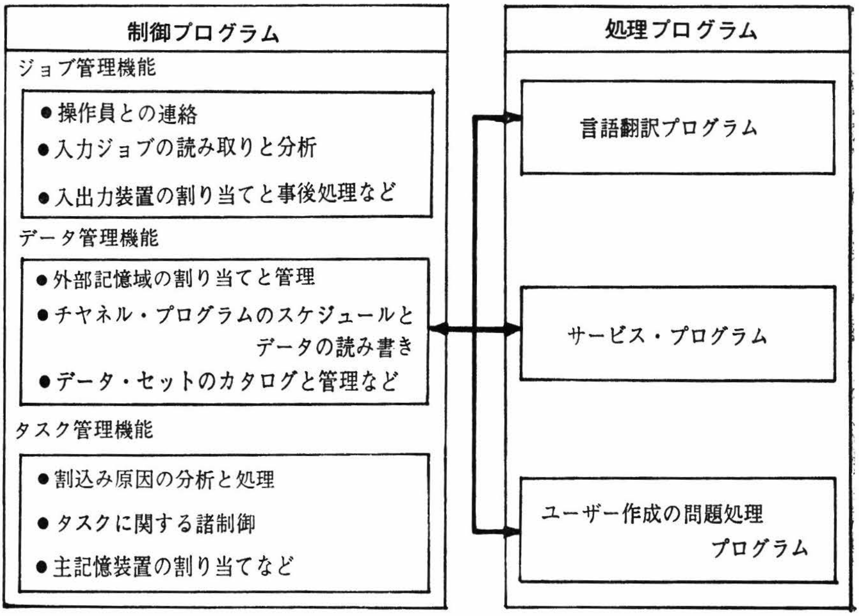
あてられると，そのジョブは，オペレーティング・システムの中では，遂行すべきタスクとしての取扱いをうける．このように，同じ単位の仕事が，システムの外と内では異った観点でとらえるのである．つまり，システム外の利用者の立場からはジョブと定義したものが，オペレーティング・システムの管理のもとではタスクとなるのである．ジョブがシステム利用者の立場での仕事の単位であるのに対して，タスクはシステム内での管理または制御の単位である．つまり，システム内ではタスクに各種資源をうまく割り振ることによってシステムの生産性の向上を保持しつつ所期のデータ処理を遂行してゆくのである．ところで，システム内に唯一のタスクしか存在しないとしたら，そのタスクに対するシステム資源の割り振りは極めて単純なものとなる．CPU にしろ，主記憶装置にしろ必要なだけ，そのタスクに占有させればよいのであるから．しかし，システム資源を無駄なくこき使うというがめつさの立場に立った場合，システム内に出来るだけ多数のタスクを同時に存在させ，それらタスクにうまく必要な資源を割り当てながらシステム全体を管理してゆくことによって，本来の性能に近い処理能力を引き出してゆかざるを得ない．いわゆる，**多重タスク・オペレーション**（multi-task operation）によるシステムの生産性の向上である．

ジョブの段階にしろ，タスクの段階にしろ何らかの形で外部媒体に記録されている諸データとのかかわりをもつし，また何らかの形式で外部媒体に処理情報を出力することを伴う．ここに，第3の管理すべき要素，つまり，データが出現する．一般にデータとプログラムは対比して考えるのが普通である．プログラムは，データ処理を指示する命令群からなっており，逆にデータは，プログラムを構成する命令によって読みとられ，処理が行われ，出力される性質のものであることから，プログラムとデータが対比して考えられるのは極めて自然なことといえよう．しかし，少し観点を変えてみると，データもプログラムも本質的に変りない．外部記憶装置上に記憶されている状態では，両者の区別はつかないし，主記憶装置への読み込みや，逆に主記憶装置から外部記憶装置に書き出す場合もそうである．異なるのは，中央演算処理装置内での機能の仕方だけである．プログラムは，処理の働きかけをするのに対して，データはそ

の働きかけを受けるという機能上の差がある．機能上の差というよりは，状態の差といった方がよいかも知れない．プログラムといえども，完全にデータとしての取扱いを受けることもあるからだ．たとえば，プログラムのある一部の命令に修正が加えられる場合がそうだ．その命令は，修正が加えられる時点では完全にデータとしての扱いをうけているのである．論旨をもとにもどそう．外部媒体とのやりとりにさいしては，やりとりの対象になるのが本来的データであっても，本来的プログラムであっても，その間に何らの差異はない．したがって，オペレーティング・システムのもとで管理される第3の要素であるデータとは，プログラムをも含んだ広義的データである．

オペレーティング・システムは，コンピューティング・システムを統一的に管理することによってその生産性の向上を計る．したがって，オペレーティング・システムには，各種処理プログラムも包含されることは前にふれた通りである．制御プログラムと呼ばれる一連の管理プログラムの指揮下で，生産的機能を果たしてゆくのが処理プログラムである．制御プログラムと処理プログラムは相互に有機的に機能し合いながら，目的とするデータ処理を達成してゆくのである．ところで，制御プログラムは，それらを構成するプログラム群を機能面から大別すると，ジョブ管理に関する機能，タスク管理に関する機能，データ管理に関する機能に大別される（**第1図参照**）．この3大管理機能が相互に補完し合いながら，コンピューティング・システムの生産性の向上に貢献するように設計されているのが制御プログラムである．

ジョブ管理（job management）の機能は，「オペレーティング・システムの概念的構造」の章でもふれたようにジョブの連続的処理を遂行することを主要な機能としている．一連のジョブに関する記述指定の行われた入力ジョブの流れを読みとり，その指定事項を解読分析したうえで，その内容を一定の形式にまとめ上げたうえで所定のテーブル（SYSJOBQE という．「ジョブ管理の諸機能とその構造」の章にて詳述予定）に書き出す．テーブルを完成したのち，実行すべき特定のジョブ・ステップに必要な入出力装置の割り当てとそのジョブ・ステップで利用される処理プログラムを呼び出す．そのあとで，管理の制御権はタスク管理プログラムに移され，1つのタスクとして実行される．タス



第1図　制御プログラムの三大管理機能

クの実行が完了すると再びジョブ管理プログラムが機能する．つまり，そのジョブに割り当てた入出力装置を他のジョブのために解放すると同時に，そのステップで作成した，あるいは使用したデータ・セットの後始末を行ったうえでつぎのジョブ・ステップまたはジョブの実行の準備にかかる．このようにしてジョブの連続処理のために機能しながら，必要なメッセージを操作員に打ち出し，操作員からの指令を分析し必要な処理の指示を行うのもジョブ管理の機能である．

一方，**タスク管理**（task management）の機能は，実行すべき仕事の単位（タスク）を順次方式または優先順位にもとづいて監視してゆくことにある．具体的には，タスクを管理するための**制御ブロック**（Task Control Block という．略して TCB）をタスクごとに作り上げ，タスクの稼動可能，待ち，異常終止などの状態の情報を維持しながら，タスクの進行状況を監視してゆく．割込みが生じると，その最初の段階で割込み原因を分析し，その原因を処理するにふさわしい割込み処理ルーチンに制御権を渡す一方，割込み処理が完了す

ると，稼動可能状態にあるタスクのうちで最も優先順位の高いタスクの処理プログラムに制御権をもどす機能を果すのもタスク管理の機能である．またタスクが必要とする主記憶装置の割り当てを行ったり，プログラム・ライブラリーから所要のプログラムを呼び込むのもタスク管理の機能である．さらに，オーバレイ構造のプログラムの実行を監視したり，タイマーに関する管理も行う．

さて，本章で詳述してゆく**データ管理**（data management）の機能であるが，その主要なねらいは，システムが取扱うさまざまなデータとデータ・セットを標準的な手法で統一的に管理してゆくことである．ジョブ管理とタスク管理の過程で発生するデータおよびデータ・セットの取扱いに関しての管理を統一的に行ってゆくのがまさにデータ管理の機能である．つまり，ジョブおよびタスクの管理において，必要に応じてデータの記憶と取り出しを行わなければならないが，データの編成方式や記憶方式に関係なく，すべてのデータの記憶，取り出しおよび更新維持を統一的に行うのが，データ管理である．

二. データ管理の背景

データの管理という観点からみた場合，今日の代表的なコンピューティング・システムは2つの問題に直面している．その1つは，データを貯えるための入出力装置の多様化の問題であり，他の1つは，取扱うデータの急増と集中化の問題である．この2つの側面は，必然的にデータの管理の仕方に新しい概念と手法を採り入れてゆくことを要請する理由となっている．その結果，オペレーティング・システムの制御プログラムには，過去のシステムでは予想されなかったような新しい機能ともいうべきデータ管理の手法を導入している．まずは，その背景から言及してゆくこととする．

1. 入出力装置の多様化と装置からの独立性

現在のコンピューティング・システムは，その利用法の拡大と高度複雑化のために，ぼう大な量のデータとプログラムを自己の支配下に置かざるを得なくなって来ている．そのうえ，技術の革新は目ざましい勢いで進歩しており，入

出力装置や補助記憶装置の種類はますます多様化し，その変化の度合いも速くなる一方である．この技術の革新が単にハードウェア構成の範囲内で対処しきれるものなら何ら問題は生じないのであるが，実際には，ハードウェアの革新は，とりもなおさずソフトウェアの革新を意味する．新しいハードウェアの出現やハードウェア機能の追加は，必然的にそれを補うためのソフトウェアの改善を促すことに直結する．ところが，新しい入出力装置の出現や入出力装置への新しい機能の追加によって，その都度，変化の影響がおよぶ処理プログラムに変更を加えることになると，プログラムの本数が増加している現在，とうてい対拠しきれないことである．そこで，その影響を出来るだけ制御プログラムの範囲内でくいとめ，処理プログラムに影響をおよぼすことを少なくするような設計理念が考え出されることになる．つまり，“入出力”に関する処理過程を新しい観点からながめ直すことによって，これまでの入出力オペレーションとは異った概念で処理してゆくことになる．

従来のデータ管理システムともいうべき**入出力制御システム**（Input/Output Control System，略して IOCS）が果してきた機能は，入出力データの転送とラベルの処理の範囲に限られていたのであるが，現在での入出力の過程が要求しているのは，データの識別，記憶，検査および取出しという統一的な機能を，単に普通のデータに対してだけでなくプログラムに対しても行うといった新しい概念でのデータ管理システムである．しかも，入出力装置の多様化という観点からすると，データ管理の機能は，直接アクセス記憶装置および順次アクセス記憶装置の双方に及ぶというだけでなく，理想的には，ある要求の特定の機能上の指定に合致しさえすれば，入出力装置が何んであれその要求を処理し得るようになっていることが必要である．そこで，データ管理の機能として，データやプログラムを識別したり，記憶したり，あるいは取出しを行うためには，プログラムのコーディングのさいには個々の特定の入出力装置を指定するのではなく，装置の種類を基準に大まかな指定を行う．実際にどの特定装置を使用するかは，ジョブ・ステップが実行される時にジョブ制御カードを通じて初めて指定する．このように実行段階になって使用する入出力装置を指定することによって，入出力装置の多様化と変化に対処するという考えが，**装置**

からの独立性（device independence）である．つまり，プログラム・コーディングにさいしては，特定入出力装置に拘泥することなく独立して行えるのであり，出来上ったプログラム自体も，実行段階まで特定装置との明確なかかわりはないのである．

装置からの独立性という機能は，入出力装置の多様化や変化に対して，プログラムの書き直しや翻訳というわずらわしさを回避しつつ，即応し得るという利点のほかにつぎのような利点も見逃すことは出来ない．

- プログラム作成のさいには，実際に使用される入出力装置に関する知識を持たなくても，プログラムの設計やコーディングあるいはデバックの仕事ができる．
- プログラムの実行効率を最適化する目的のために，さまざまな入出力装置の組合せやバッファリングの手法を試してみることができる．なぜなら，プログラムで利用する入出力装置やバッファリング手法の指定はプログラムの作成時でなく，実行時に選択指示できるから，どの組合わせが一番効率的かを思いのままの指定でテストすることができるからである．
- 異なったコンピューティング・システムの異なった入出力装置の構成をもつシステムでも実行できるようなプログラムを作り上げることができる．このことは，複数台のコンピューティング・システムを導入しているユーザーにとっては，極めて有利となる点である．
- 出力レコードのブロック化因数の指定も実行時までのばすことができることから，処理効率を最大にするようにブロック化因数を変えることができる．

ところで，このような装置からの独立性という機能は，どのような手法で実現可能になっているのであろうか．その詳細はあとにゆずるとして，ごく基本的な考え方だけをここで概観しておくこととしよう．従来の IOCS の考え方では，与えられた仕事の処理に必要な入出力ルーチンを，IOCS のマクロ指定にしたがって言語翻訳の段階でユーザーのプログラムの中に組入れるというやり方である．この方法を用いれば，プログラムの翻訳後に入出力ルーチンを組入れて編集したりする必要がなく，アセンブリーだけで実行可能な完結したプロ

グラムが得られるという利点がある．しかし，この方法には重大な欠陥もある．この方法では言語翻訳の過程ですべてを完全にする必要があることから，必然的に原始プログラムの中に入出力装置の種類やそれにかかわりをもつオペレーション・モードに関する詳細な指定をしておく必要がある．当然のことながら，入出力に関する細かな働きに注意を向けなければならないし，適用業務や機械構成上で入出力の機能に影響をもたらすような変化が生じると，最初からプログラムの手直しを迫られることになる．変化に十分対拠しきれないのが，従来のデータ管理の機能である．

新しいデータ管理にあっては，多重タスク・オペレーションという複雑な処理手法に加えて，これまでに見られなかったような入出力装置の多様化と変化に対処してゆくため，従来のそれとは異った考え方を採用せざるを得なくなった．つまり，データ管理に関係する種々の機能を果すルーチンは，プログラムの翻訳段階やプログラムのローディング段階に組込まれるのではなく，プログラムの実行そのものの過程の中で必要が生じるごとに，必要な機能が組入れられて働くような考え方に立っている．実際には，OPEN マクロ命令が実行されることにより，必要な機能を果すデータ・アクセス用のルーチンが組込まれ機能するようになっている．このデータ・アクセス用のルーチンは，再入可能という属性をもたされたルーチンであるので，同じような要求をもった複数個の異ったタスクが，同じアクセス・ルーチンを共通的に併用することもできる．したがって，OPEN マクロ命令が実行された時，すでに所要のアクセス・ルーチンが主記憶装置に存在しているならば，ライブラリーからの呼び込みは行なわれず，そのルーチンが該当タスクに対しても機能するような前準備が行なわれるだけである．

ともあれ，プログラマーは，プログラム作成時には，利用できる**アクセス方式**（access method）のうちの必要なものを選んで適切なマクロ命令で指定し入出力オペレーションを要求する．入出力装置の種類だとかバッファリング手法，チャネルの使用の仕方やデータの諸特性などは，実行段階になって入力ジョブの流れの中で“データ規定（DD）ステートメント”を使って指定できる．このように実際にプログラムが実行される時点になって，データ処理上の

性質とかデータの諸特性が指定できるのは，プログラムの実行過程でアクセス・ルーチンが組込まれ，機能するような設計思想に立脚しているからである．これこそが，“装置からの独立性”という機能を実現するための基本手法なのである．

2. データの急増と統一的管理の必要性

過去数年の間に，代表的な企業では会計処理，人事処理，財務処理あるいは販売処理などの大半をコンピュータ処理化してしまった．その結果，企業活動に必要な基本的データのほとんどが，一台または数台のコンピュータに集中的に蓄積されてきた．1たん穿孔カードや紙テープに記録されたデータは，さらに集約化された形式で磁気テープ，磁気ディスクあるいは磁気ドラムといった媒体に移しかえられることによって，データを記憶保管するに要するスペースを減少させることはできたが，一方，処理の対象になるデータ量は爆発的な速度で増加の一途をたどっているので，そのデータを有効に管理し，効果的に活用することさえ困難な状況に追込まれているのが実情である．コンピュータに集中した多量の情報を効果的に管理する方法が欠如していた結果，非常に沢山のデータが重複して存在することにもなっている．たとえば，ほとんど同じような内容をもったデータ群が全く別々に生成され，記憶，保持されているケースもまれではなくなっている．このような状態は，データを最新なものに更新するさいにも，またデータを管理してゆくさいにも非常にむずかしい問題を投げかけることになっている．データ利用の効果的管理の方法が採用されないかぎり，ますます個別的にデータを収集，保持をするようになり，そのデータを統合化し共有してゆくという方向には考えも及ばなくなってしまう．そのうえ，新しいデータが急増する一方では，不要になったデータの多くがそのまま蓄積されるままになっている．その結果，コンピュータ部門の要員は，自分に任された多量のデータにどのように対処すべきかとまどってしまっているのが実情である．磁気的に記録されたデータは知覚することが出来ないせいもあって，止むを得ず手作業で入念な記録をとることによって，データを分類し，索引を作り上げ，あるいは必要なデータをつきとめる方法を考えている．またデ

ータの利用の仕方を管理したり，利用後の後始末に関しても同様であるし，データを貯えるためのスペースを見つけだして割り当てるさいにも同様に手作業で行っている．このような方法だと，エラーも生じやすいし，要員の移動によって生じる問題などもある．

このように，コンピューティング・システムへのデータの集中化と急増は，厄介なデータ管理上の問題点を提起している一方，データ管理上都合の良いことにもつながる．データが集中化されてくると，必然的に企業の日々の活動に必要となる重要な記録のほとんどすべてが1つの個所に集まることを意味しており，それらのデータを1つの情報体系として結合できることを示唆している．つまり，数学的あるいはプログラミング上の手法を駆使すれば，この多量の新しい情報体系の中から経営管理上に必要となる報告書とか統計的データを引き出してゆくことが可能となるからだ．言いかえれば，過去の伝統的な処理で必要としていたと同じデータを有機的に統合化することによって，新たな創造的手法での**データ・ベース**（data base）として十分使いうるのである．

さて，コンピュータ部門へのデータの集中と急増が投げかけた問題を解決し，それによってもたらされたチャンスを有効に活用するためには，統一的なデータの管理体系こそが重要となる．とはいうものの，コンピュータの導入先によって，あるいはコンピュータで処理させる適用業務によって，データの管理に要求される条件は異ってこざるを得ないので，すべての利用先のデータ管理上の問題点を解決し得るような体系を見出すことはむずかしい．しかし，各々の利用先が，自己の特殊なニーズに見合った効率的なデータ管理の体系を作り出ことを可能にするような各種便宜をオペレーティング・システムの機能の一部として提供することはできよう．

このようなデータの急増と集中化という環境を考慮して，データ管理の諸機能として組込まれているのが，つぎのものである．

- データの編成に関して，一貫した手法で行えるような機能．
- データ・セットの所在をつきとめるのに利用できるカタログの機能．
- データの記憶や取り出しのためのいくつかの手法．

これら機能を通して，標準的かつ統一的なデータ管理が行えるように工夫さ

れている．それがどのような手法で具体的に実現できるかは，後述することとする．

三. データ管理に関する概念的機能構造

データを統一的な手法で管理してゆくための基本的要件は，言うまでもなく一貫性のある方法でデータの編成を行うことにある．そうでないことには，広範囲にわたる利用者が容易にデータの共有を計ることができないばかりか，処理プログラムの実行を制御プログラムで統一的にコントロールしてゆくことも困難になってしまう．そこで一貫性のあるデータ編成を目指さざるを得ないわけであるが，実は，制御プログラムの行うデータの編成方法は，多くの点で伝統的に行ってきた方法を踏襲しているに過ぎない．

制御プログラムがかかわりをもつデータの最少単位は，通常レコードである．**レコード**（record）は，御承知の通り連続した複数個のフィールドから構成されるわけであり，一般には相互に関連したデータの集りである．どんな場合にも，制御プログラム自体は，フィールド単位でデータを取出したり，フィールドそのものを認知することはできない．制御プログラムがアクセスできるのは，レコードが最少単位であり，フィールドの取扱いは処理プログラムの責任とならざるを得ない．

1つまたは2つ以上のレコードが物理的に連続した形式で記録されたものが**ブロック**（block）である．補助記憶装置に記憶されたブロックは，**ギャップ**と呼ばれるもので他のブロックと分離されるのが普通であり，かつ，主記憶装置との間での転送の単位になるのもブロックである．レコードが，論理的に一体をなしているということから複数個のものをグループ化してブロックとすることもあるし，単に補助記憶装置のスペースを効率的に使うとか主記憶装置と補助記憶装置間でのデータ転送の時間的効率を高める目的でブロック化することもある．

オペレーティング・システムは，全てのレコードの桁数が一定であろうと，それぞれのレコードの桁数が異っている場合であろうと，そのレコードを記憶

したり取り出したりすることができる．レコードの桁数が異る場合，つまり**可変長レコード**（variable length record）の場合には，個々のレコードの桁数が何桁であるかを各レコードの最初のフィールドで明示することによって，レコードの転送を制御する．同様にレコードがブロック化されている場合には，各々のレコードの桁数やブロックに収容されるレコード数が異ることから，各ブロックの先頭に位置するレコードの最初のフィールドにブロック全体の長さを示すことによって，ブロックの転送を統一的に制御する．つまり，可変長の場合の各レコードの長さとブロックの長さを示す2つのフィールドは，ブロックから特定レコードを引き出すために制御プログラムの一部であるアクセス方式のルーチンが利用するのである．ブロックやレコードの長さが定義されていない可変長レコードの場合であっても，ブロック単位での記憶や取出しを制御プログラムで行うことはできるが，その中の特定レコードを取り出して処理するとか，ブロッキングを行う作業は処理プログラムに委ねざるを得なくなる．ブロックやレコードの長さが明示されているか**固定長レコード**（fixed length record）の編成の場合にだけ，制御プログラムが統一的手法でブロッキングやディブロッキングの面倒をみてくれるわけである．

さて，オペレーティング・システムはこのようにして作り上げられたさまざまなデータの集合体を取扱うことになる．たとえば，原始プログラムの集合体，目的プログラムやロード・モジュールの集合体，メッセージの待行列，統計データのテーブル，各種データ・ファイルなどがそれである．これらのいずれも互に関連性をもったレコードないしはブロックの集合体である．オペレーティング・システムでは，このようなレコードの集合体を特に**データ・セット**（data set）と呼んでいる．データ・セットは，単に相互に論理的に関連するデータの集合体というよりは，特別な名前がつけられたデータの集合体とみなされる．別な表現でいえば，データ・セットは，その名前，外部記憶装置で占める場所，およびデータ・セットの性質を記述するパラメータによって定義されたデータの集合体である．これらデータ・セットを定義する情報は，データ・セット・ラベルに貯えられており，通常データ・セット自体が貯えられると同じボリューム上に収容される．実は，制御プログラムがデータ・セットを管理

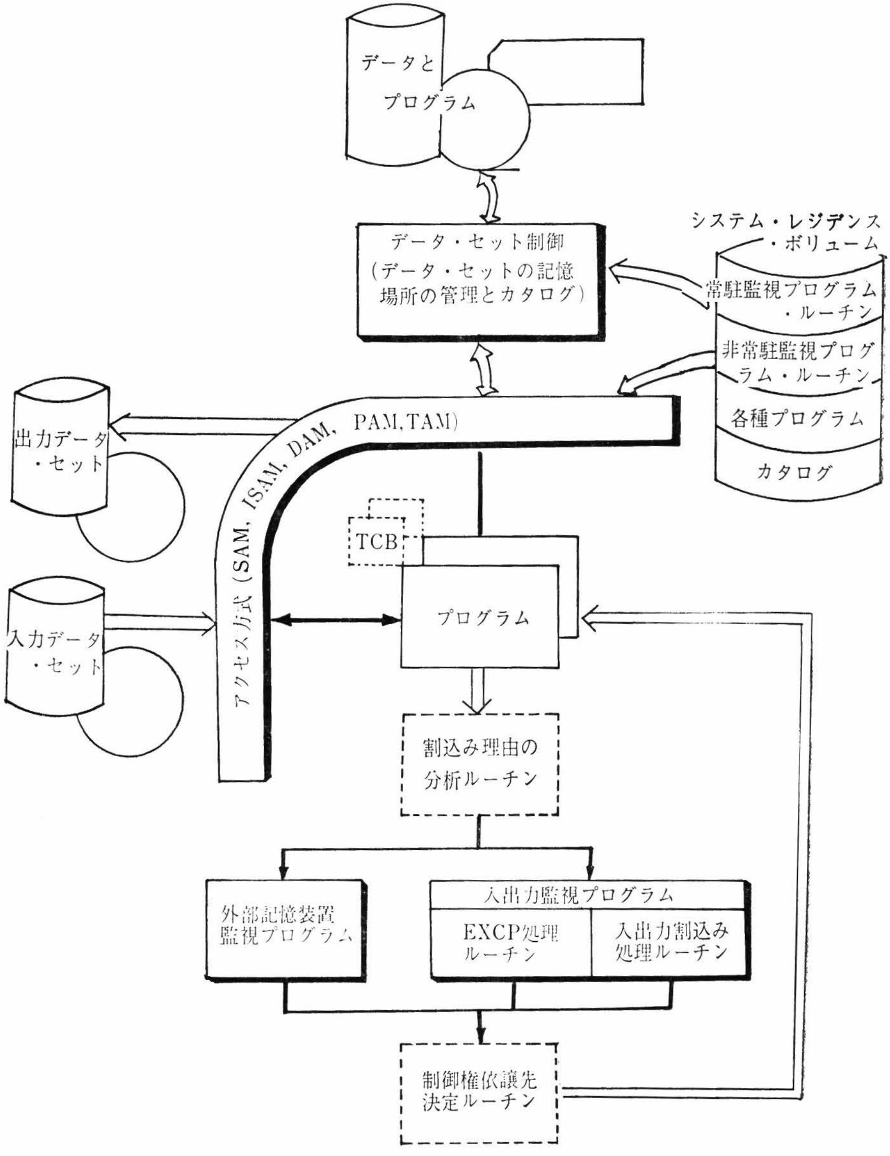
するにあたっての重要な情報源の１つがこのデータ・セット・ラベルなのである．

ここで，前述した“ボリューム”に関して正しい定義を試みておこう．

ボリューム（volume）とは，一つの読み書きの機構で，他の読み書き機構とは独立してオペレーションのできる補助記憶装置のスペース単位である．このボリュームは，補助記憶装置の標準的単位ともいえるのである．具体的には，テープ・リール，ディスク・パック，データ・セルあるいはドラムが１つのボリュームといえる．しかし，特殊な磁気ディスク記憶装置のように，何組かの読み書き機構が独立的にオペレーションを行いうるようになっている場合には１つの機構でアクセスし得る記憶域がボリュームと定義される．このボリュームも，制御プログラムがデータを管理する場合の重要な単位となる．制御プログラムがボリュームの識別に使うのが，ボリューム・ラベルであり，これはデータ・セットの識別に使われるデータ・セット・ラベルに対応する．ボリューム・ラベルには，必ずボリュームの一貫番号を示す情報が貯えられる．直接アクセス記憶装置の場合には，ボリューム・ラベルに当該ボリュームに収容されているデータ・セットのラベルを入れてある場所（Volume Label of Content, 略して VTOC と呼ばれる）を示す情報も貯えられる．つまり，こうすることによって，ボリュームの番号とデータ・セットに付けられた名前だけが与えられると，データ管理を行う制御プログラムは，まずボリューム番号を手がかりに該当ボリュームをボリューム・ラベルを通じて捜し出す．つぎにボリュームが見つかれば，あとはボリューム・ラベルに入っている VTOC を示す情報をもとに該当するデータ・セット・ラベルをつきとめることができ，このラベルからデータ・セット自身をアクセスするに必要な情報を得ることができるのである．

これまでデータ編成にかかわりをもついくつかの用語の定義と概念についてふれてきたのは，これから説明しようとするデータ管理に関する機能構造を説明するうえでの前準備として回避することができなかったからである．

さて，データ管理の基本的機能構造であるが，その概略は**第２図**に示した通りである．その主要機能は次の３つに大別される．



第2図　データ管理に関する機能構造

- データ・セットの識別を行いその所在をつきとめるためのラベルとカタログにかかわりをもつデータ・セットの制御の機能．
- データ・セットの編成やデータの記憶と取出しを標準的手法で行うためのデータ・アクセスの機能．
- 実際的入出力装置の制御と入出力オペレーションによって生じるエラー処理の面倒をみる入出力監視プログラムの機能．

なお，前節でふれた“装置からの独立性”という機能は上記3つの主要機能がうまくからみあうことによって，達成されているものである．以下に3つの主要機能に大別して，その働きを概観してゆくことにしよう．

1. データ・セットとデータ・セットの制御機能

1）データ・セットの特性

互いに論理的関係をもったレコードに名前を付けて編成した集合体をデータ・セットとして取扱うことは前にふれた通りである．データ・セットに収容される情報は，特殊な形式，目的あるいは記憶媒体によって制約されるといったものではない．たとえば，原始プログラムであってもよいし，マクロ命令のライブラリーであっても処理プログラムでプロセスされるデータ・ファイルであってもかまわない．制御プログラムからすれば，名前の付けられたデータの集まりであることで十分なのである．データ・セットに名前を付けるという考え方は，オペレーティング・システムでデータ管理の機能を果すうえでの最も重要な概念の1つである．これがデータ・セットをカタログし，データ・セットの名前に修飾語をつけて管理してゆくことの基本的発想となっているからである．データ・セットの名前は，単純な名前であってもよいし，修飾語で段階を表わすように意図した単純名の連続であってもよい．たとえば，HANBAI. TOKEI. SALESMAN というデータ・セット名は，3つの単純名を結合したものから成り立っており，段階的区分を体系的に表現したものである．つまり，販売管理関係のデータ・ファイルのうちの統計的資料に関係するもので，セールスマン別にまとめられたデータ・ファイルということを表わしている．先頭の単純名が大区分を，つぎの単純名が中区分を，そして，最後の単純名が

小区分をといった具合いに，データ・セットを体系的に整理保管するのにデータ・セット名は好都合であるように工夫できるのである．データ・セットが作成され，外部記憶装置に貯えられたうえで名前が付けられると，後述するようにこの名前がカタログ・システムの一部となり，そのあとは名前だけでデータを検索し取り出すことが可能になるのである．

2）データ・セット・ラベルの利用

前述したようにジョブ制御ステートメントの1つであるDDステートメントを使って，データ・セットのさまざまな性質を制御プログラムに知らせることができる．このステートメントを使って，データ・セットが必要とするスペースだとか，ボリュームの種類あるいはデータ・セットの名前などを指示することができるのだ．ジョブ・ステップが開始すると，制御プログラムは必要なスペースをデータ・セットに割り当てると同時に**データ・セット・ラベル**（data set label）を作り出す．作り出されたラベルの内容が完成するのはプログラムの実行中に OPEN あるいは CLOSE マクロ命令が処理された時である．これが磁気テープの場合だと若干異る．磁気テープの各リールがそれぞれ1つのボリュームと見なされることはいうまでもない．磁気テープはレコードが記録されている順番にしか処理できないという性質をもっていることから，ボリュームやデータ・セットの識別方法が直接アクセス記憶装置の場合と幾分異ってくるのは当然である．直接アクセス記憶装置の場合だと，ボリューム・ラベルの中でデータ・セット・ラベルが一括して収容されている VTOC の場所を指してくれているのでそれを手がかりに該当データ・セットを識別することができた．磁気テープの場合にも，標準的な手順としては，ボリューム・ラベルとデータ・セット・ラベルを用いるが，各データ・セット・ラベルは2つの部分から構成されている．データ・セットの直前に記録される**ヘッダー・ラベル**（header label）とデータ・セットの最後に記録される**トレーラー・ラベル**（trailer label）の中に各種のデータ・セットに関する情報が貯えられる．テープ・ボリュームのどれに該当データ・セットが収容されているかは，ボリューム番号によって指示され，容易に捜し出すことができる．

制御プログラムの中には標準化されたテープ・ラベルを処理する手順は用意されているが，ユーザーが希望する好みのラベルの処理に関しては，ユーザーが自分で作成しチェックするルーチンを組み入れる必要がある．また，ラベルを全く持たない磁気テープの使用も許されるが，この場合には，オペレーターが適切なボリュームを所定の装置にあらかじめセットする必要が生じる．制御プログラムが，適切なボリュームを選択するとか，適切なボリュームをセットするように指示するための何らの手がかりをもたないからである．

3）カタログと制御ボリュームの機能

データ・セットの数が増えてくると，システムに収容されているデータ・セットの在庫目録（カタログ）を絶えず最新の状態に更新し管理してゆくことは至極やっかいな仕事となる．このような仕事をプログラマーやオペレーターから解放し統一的に管理するための機能が，データ・セットの**カタログ・システム**（catalog system）である．このカタログは，直接アクセス記憶装置内に収容され，樹木式構造をとる一連のインデックスから構成される．それぞれの導入システムの要求に合致させるために，この樹木式構造をどのように組立てるかはすべて使用者の立場にまかせられている．実は，このカタログに登録される名前こそが，データ・セットの名前なのである．したがって，データ・セット名に付けられる各々の修飾名が樹木式構造の段階を示すインデックスに対応する．たとえば，マスター・インデックスの中では HANBAI という項目を，2番目の段階のインデックスでは TOKEI という項目を，3番目のそれでは SALESMAN という項目をそれぞれ捜し当ててゆくと，HANBAI. TOKEI. SALESMAN という名前のデータ・セットが見付かる．最後の段階のインデックス，すなわち SALESMAN という項目の1部には，このデータ・セットを収容しているボリューム番号と装置の種類を識別する情報が貯えられている．さらに磁気テープのような順次アクセス装置の場合だと，そのボリュームの何番目のデータ・セットとして当該データ・セットが貯えられているかを示す順序番号も一緒に貯えられている．それゆえに，そのボリュームに複数個のデータ・セットが収容されていたとしても，容易に該当のデータ・セットのあ

る場所までたどりつくことができるわけである．

カタログの全体または一部分を保管しているボリュームのことを特に**制御ボリューム**（control volume）という．通常，システム・レジデンス・ボリュームの一部はカタログを収容する目的で使われるので，制御ボリュームといえる．しかし，システム・レジデンス・ボリュームに全てのカタログを保持する必要はない．お互に関連のある幾つかのデータ・セットのグループごとに別々の制御ボリュームを適用することは意味がある．カタログのうちであるデータ・セット・グループに関係する部分だけが自由に移動することができるからでる．制御ボリュームのこのような使い方は，あるグループのデータ・セットを通常使用するコンピュータから別のコンピュータに移して使用するという可能性が考えられるような場合特に重要なことになる．

すでに登録されているデータ・セットを捜し出す作業を制御プログラムが行うさいには，まずシステム・レジデンス・ボリュームを起点として開始し，インデックスを一段階づつ下げてゆき，求めるデータ・セットの入っているボリューム番号が得られるまで続けられる．求めるボリュームがまだ入出力装置にセットされていないことが判明すると，その旨のメッセージをオペレータに出す．求めるデータ・セットが直接アクセス記憶装置に貯えられている場合にはそのデータ・セットが貯えられている場所を捜し出す作業は，指示されたボリュームのボリューム・ラベルを手がかりとして，データ・セット・ラベルが収容されている VTOC のありかを見つけ出すことによって行われる．一方求めるデータ・セットが磁気テープ装置に入っている場合だと，最下位のインデックスから得られた順序番号をもとに該当データ・セットを探し出すわけだ．

なお，定期的に更新されるデータ・セット，あるいはデータ・セット群の一部分がそれぞれ異った時点に作られるといったデータ・セッットに名前をつける特別な方法も設けられている．このような場合，1つのデータ・セットを更新してできた各データ・セットは**世代**（generation）と呼ばれ，それぞれのデータ・セットにかかわりをもたせる番号を**世代番号**（generation number）と呼ぶ．そして古い順に並んだデータ・セットをひとまとめにして**世代別データ群**（generation data group）といい，同じデータ・セット名で参照できるよ

うになる．その場合，特定世代のデータ・セットだけを取り出すには，絶対的世代名あるいは相対的世代名のいずれかを使うことによって可能となる．詳しくは，後述する．

4）データ・セットの保護機能

コンピュータに広範囲にわたるデータが集中されてくると，特定の人にしか解放することができない機密情報が組込まれてくるし，また重要なものは破壊や盗用から保護する必要のある情報も増えてくる．そこで，データ・セットの機密保護の機能を設けることによって，特定の利用資格のある人だけがそのデータ・セットのアクセスができるようにしている．データ・セットを保護するためには，そのデータ・セットは標準ラベルをもつ磁気テープ・ボリュームか直接アクセス記憶装置に収容されている必要がある．保護されていることを示すためにラベル内の標識はONにセットされる．この保護用の標識はデータ・セットがOPENされるさいに検査される．その結果，標識がONであると，正しく符号する**合言葉**（パスワード）をコンソールから入れてやらなければならない．この合言葉は資格ある人にだけ知らされていることが前提となるが，もし正しい合言葉が入力されないとOPENルーチンは処理の続行をやめてしまうので，そのデータ・セットを保護することができる．このようにして，データ・セットを保護することもデータ管理の機能である．

2. データ・セットの編成とアクセス方式

データの記憶と取り出しを決定する第1の要素は，データ・セットの編成の方法であり，第2の要素はデータのアクセス技法である．この両者の組合わせによって，入出力データ・セットの**アクセス方式**（access method）が決定される．そのうえで，第2図に示したように該当するアクセス方式のルーチンを通じて入出力データ・セットの作成，更新，データの取り出しが行われるのである．これらアクセス方式のルーチンは，処理プログラムに組み込まれるというよりは，独立したルーチンとして制御プログラムの一部を構成し，処理プログラムにデータの取り出しや記録のサービスを提供するのである．

オペレーティング・システムでは，5つのデータ編成法が標準的なものとして用意されており，この範囲の編成法であるならば，データ管理の機能によって容易に作成・保持ができるようになっている．5つの編成法の概要はつぎの通りである．

- **順次編成データ・セット**（sequential data set）：データ・セットを構成するレコードが論理的順序というよりは物理的順序で記録されるような編成法であり，物理的な順序でレコードがアクセスされるような記憶媒体（磁気テープ，紙テープ，穿孔カードあるいは印刷用紙）に収容されるレコードから成るデータ・セットがこれに当る．したがって，ある1つのレコードが取り出されると，つぎのレコードの位置は，必然的にそのデータ・セット上の物理的位置によって決定されるといった性格のものである．一般的には，すべての先行するレコードの読み書きが終了しないかぎり，つぎのレコードの読み書きは行えないのが原則である．また，レコードの更新，追加削除といった仕事は，データ・セットをコピーする方法でしか行えない．なお，直接アクセス記憶装置に収容するデータ・セットでも，必要なら順次編成をとらせることもできる．
- **索引順次編成のデータ・セット**（indexed sequential data set）：この編成では，データ・セットを構成するすべてのレコードがキーの順番に配列される．キーとは，レコード内の主要な制御番号のことである．さらに，各レコード記憶位置を探し当てるためのインデックスがいくつか作り出され保持される．したがって，この編成ではレコードを順次にアクセスできるだけでなく，任意の順序でアクセスすることもできる．なお，この編成法はその性質からいって直接アクセス記憶装置にだけしか適用されない．
- **直接アクセス編成のデータ・セット**（direct access data set）：この編成法も，直接アクセス記憶装置にしか適用されない．これは，索引順次のデータ・セットの編成と若干似てはいるがインデックス・テーブルは必要とせず，かつレコードをどのような方法でアクセスするかは完全にプログラマーに任されている．プログラマーが指定したアドレシングの手法でレコードを直接的に記憶したり，取り出したりできる編成法である．特定レコ

ードを指示するためのアドレスは，ハードウェアに付けられた実際番地であってもよいし，データ・セットの最初のレコードを基準にした相対的番地であってもよい．

- **区分編成のデータ・セット**（partitioned data set）：この編成法は，順次編成と索引順次編成の双方の特徴を兼ね備えたものである．メンバーと呼ばれる順次編成された独立のデータ・グループが直接アクセス記憶装置に貯えられる．各メンバーには単純名が与えられ，データ・セットの一部を構成する**登録簿**（directory）に登録される．この登録簿をみると，そのメンバーがどの場所に記憶されているかが分るのである．1つのデータ・セットには，多数のメンバーを収容することができるし，また1つのメンバーは1つ以上の順次編成のブロックで構成できる．この区分編成法は，一般にはプログラム群を収容するのに使われる．通常，**プログラム・ライブラリー**と呼ばれるのがこの編成法であり，個々のプログラムがメンバーである．登録簿は，メンバー名とその別名の一覧表であり，各メンバーの収容されているアドレスの組合わせで構成されている，別名にかかわるアドレスは，メンバー内のどのレコード位置を示すものであってもよい．ともあれ，プログラム・ライブラリーに登録されているプログラムをプログラム名あるいはその別名で呼び出すことができる裏にはこのような編成法があるのだ．
- **通信編成のデータ・セット**：この編成は，遠隔地のオンライン端末から受取ったメッセージの待行列，または逆に端末へ送り出すメッセージを処理するための編成である．メッセージの待行列を作るための機能およびメッセージを待行列から取り出すための機能をからみ合わせた編成法である．なおメッセージの待行列は主記憶装置の中に作るだけでなく，直接アクセス装置にも作られる．

これらデータ・セットをアクセスする技法には，大別して2つのものがある．その1つは，待機アクセス技法と呼ばれ，他は基本アクセス技法と呼ばれる．これら技法は，バッファリング，CPU の処理と入出力オペレーションの同期化の取扱いがどうかによって区分されているのである．

待機アクセス技法 (queued accessing technique) は，できるだけプログラマーの負担を少なくして，データ管理プログラムでその分をカバーしようという意図がもたされている．レコードの取り出しを行うには，GET マクロ命令を使い，レコードを外部記憶媒体に出力するのには，PUT マクロ命令を使う．このマクロ命令を使うとバッファの自動的管理，ブロッキング／ディブロッキングおよび CPU のプロセスと入出力オペレーションの同期化など自動的に行われることにより，プログラミング上の負担が軽減する．

一方，**基本アクセス技法** (basic accessing technique) では，入出力のために READ および WRITE マクロ命令を使用する．これらマクロ命令は，物理的データ・ブロックの転送を行うだけであって，論理レコードの取扱いはしない．つまり，ブロッキング／ディブロッキングはプログラマーの責任である．この技法は，制御プログラムが次に処理すべきレコードの順番を予知しきれない編成のデータ・セットを取扱うとか，待機アクセス技法で行われる自動的機能の一部あるいは全体を使いたくないときに意味をもってくる．

以上のアクセス技法とデータ・セットの編成法の組合せを“**アクセス方式** (access method)”と呼ぶ．**第1表**に示したのが，データ・セットの編成法に対応したアクセス方式の種類である．アクセス方式の選択にあたっては，データ・セットの編成法だけでなく該当アクセス方式で提供されるマクロ命令の機能範囲に関しても考慮を払う必要がある．

データ・セットの編成方法	アクセス方式	
	QUEUED(待機)	BASIC (基本)
順次編成 (SEQUENTIAL)	QSAM	BSAM
索引順次編成 (INDEXED SEQUENTIAL)	QISAM	BISAM
直接アクセス編成 (DIRECT ACCESS)		BDAM
区分編成 (PARTITIONED)		BPAM
通信編成 (TELECOMMUNICATION)	QTAM	BTAM

第1表 データ・セットの編成方法とアクセス方式

ともあれ，データ管理プログラムは，データ・セットの編成法とアクセス技法を組合わせたアクセス方式ルーチンを使って標準的なデータの取出し，記憶をあらゆる入出力装置に対して行ってくれるのである．

3. 入出力監視プログラムの機能

入出力監視プログラムは，実際の入出力オペレーションのすべてを行う．アクセス方式のルーチンを通じて出される処理プログラムからの入出力オペレーションに対する要求を受け付け，必要な場合にはその要求を待行列に入れ，該当する入出力装置が空いた時に必要な命令を出す．また，データ・セットに割り当てられた主記憶装置の記憶域の大きさを入出力の要求する記憶域が越えてしまわないような監視も行う．

入出力オペレーションが完了すると，そのことをアクセス・ルーチンに通知すると同時に，必要なら標準的な入出力エラー訂正のルーチンを実行する．なお，アクセス用ルーチンと入出力監視プログラム間での連絡はすべて EXCP マクロ命令を用いて行われる．入出力監視プログラムの詳細な機能については「第３章オペレーティング・システムのためのハードウェア構造」の「入出力オペレーションと監視プログラム」の項で取扱っているので参照されたい．

四．ボリューム・ラベルとデータ・セット・ラベル

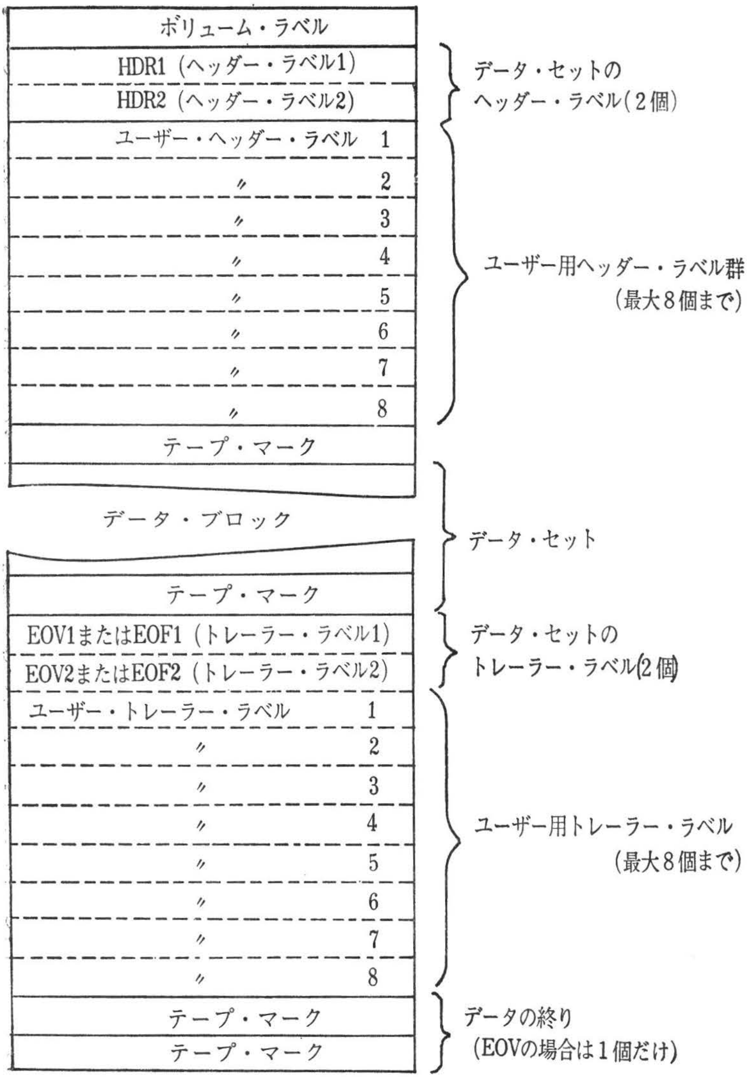
磁気テープのラベルにしろ，磁気ディスクのラベルにしろ，もともとはオペレーティング・システムとは無関係に考えだされ利用されてきた．人間の目で知覚することのできない磁気的記録であることから，間違いのないファイルを利用しているかどうかを自動的に確認する方法として，処理すべき全レコードあるいはブロックを確実に入力したかどうかをチェックする方法として，あるいは，有効期限内にあるデータ・ファイルを破壊しないことを保証する手段として活用されてきた．このラベルの利用は，磁気的記録からくる利用者の心理的不安を和らげることにも多分に貢献してきた．オペレータが，誤って間違った磁気テープをセットしてもラベルのチェックで発見されるからという安心感

が生まれたし，プログラマーは，プログラムの誤りで入力データを読み落したとしてもトレーラー・ラベルでの突き合わせでそのことが見つけだされることで不安から解放された．

オペレーティング・システムのデータ管理プログラムに先行したIOCS (Input/Output Control System) では，このラベルの処理が1つの重要な機能であった．入力ファイルの場合には，正しいファイルの収容された磁気テープがセットされているかをチェックし，入力ファイルの処理が終るとそのファイルに収められている全レコードと全ブロックが読込まれたかどうかをトレーラー・ラベルを通して確認した．一方，出力ファイルの場合には，利用する磁気テープに以前に記録された内容がすでに有効期限を過ぎたものであることを確認したうえで，ヘッダー・ラベルを書き出し，そのあとに出力レコードを書き込む．出力レコードとブロックの数を算出しておいて，ファイルの最後にトレーラー・ラベルを書き出すのも IOCS の機能であった．

ところで，オペレーティング・システムのもとでのラベルは，データ管理機能の設計思想を十分に反映したうえで作り上げられた．かってのラベルの目的を包含しながらも，新たな機能を配慮して設計されたラベルゆえ，かなり大幅な変更が加えられざるを得なかった．カタログ機能への考慮，世代別データ群への考慮，データ・セットの保護機能への考慮，装置からの独立性への配慮あるいは DASD (Direct Access Storage Device) 上のスペースの自動管理機能への考慮などが100パーセント反映された設計がなされている．したがって，これらラベルを素通りしたのでは，とうていデータ管理の機能は理解できないのである．そこで，まずはラベル体系についての言及からスタートしよう．

ラベルには大別して2種類ある．その1つは，ボリューム・ラベルであり，もう一つは，データ・セット・ラベルである．ボリュームとは，先に定義したように磁気テープ1巻，ディスク・パック1個，あるいは1つのアクセス機構でカバーし得る DASD 上の記憶域である．1つのボリュームには，複数個のデータ・セットを収容することもあるし，逆に1つのデータ・セットが複数のボリュームにおよぶこともあり得る．それゆえ，ボリュームを管理するためのラベルとデータ・セットを管理するためのラベルを区別せざるを得ないことに



第3図　標準テープ・ラベルの構成

なる．ボリュームそのものに関する属性を定義したのが**ボリューム・ラベル**であり，データ・セットにかかわりをもつ属性を定義したのが，**データ・セット・ラベル**である．

以下に磁気テープおよび DASD のボリューム・ラベルとデータ・セット・ラベルについて詳述することにする．

1.　磁気テープのボリューム・ラベルとデータ・セット・ラベル

オペレーティング・システムのもとでは，標準ラベルをもった磁気テープ・ボリューム，非標準ラベルのボリューム，ラベルをもたないボリュームのいずれも処理可能である．しかし，データ管理の機能を有効に活用するためには，言うまでもなく標準ラベルをもった磁気テープ・ボリュームであることが要求される．したがって，ここでの説明の焦点は，標準ラベルということになる．

標準テープ・ラベルは，ボリューム・ラベルと一連のデータ・セット・ラベルから構成される（**第3図参照**）．ボリューム・ラベルは，磁気テープ上の先頭のレコードであり，ボリュームの識別情報や所有者の情報を貯える目的に使われる．

データ・セット・ラベルは，磁気テープ上のデータ・セットの前後にあり，データ・セットを識別する情報とデータ・セットの性質を記述する情報が含まれている．一連のデータ・セット・ラベルは，ヘッダー・ラベル，トレーラー・ラベル，ユーザー用ヘッダー・ラベルとユーザー用トレーラー・ラベルに大別される．データ・セットに先行するのが，ヘッダー・ラベルであり，後続するのがトレーラー・ラベルである．

標準ラベルは，いずれも80桁のレコードから構成され，最初の4桁の文字で，ラベルの種類を識別する．4桁の識別名は，つぎの通りである．

1）　VOL1：ボリューム・ラベルであることを示す．

2）　HDR1 と HDR2：データ・セット・ヘッダー・ラベル1と2を示す

3）　EOV1 と EOV2：データ・セットがボリュームの終りに達したときのトレーラー・ラベル1と2を示す．この識別名をもったトレーラー・ラベルであることは，データ・セットがそのボリュームに収容しきれて

いないことを意味し，後続するデータは別のボリュームに収容されていることを暗示する．EOV は“End Of Volume”の頭文字である．

4） EOF 1 と EOF 2：データ・セットの完全な終りを示すトレーラー・ラベル 1 と 2 を示す． EOF は“End Of File”の頭文字である．

5） UHL1 から UHL8：ユーザー用ヘッダー・ラベルの 1 から 8 までを示す．ユーザー用ヘッダー・ラベルは最高 8 個まで許される．

6） UTL1 から UTL8：ユーザー用トレーラー・ラベルの 1 から 8 までを示す．ユーザー用トレーラー・ラベルは最高 8 個まで許される．

データ管理プログラムでの有効処理を行うためには，単にラベルが標準的であることだけで十分だということにはならない．さらに，**テープ・マーク**と呼ばれる 1 桁の特別なレコードが所定の個所に存在することが条件となる．このテープ・マークは，第 3 図で示した個所に位置し，データ管理プログラムに次の条件を示す．

- ヘッダー・ラベル群の終りを示すと同時に処理すべきデータ・セットの始まりを指示する．
- データ・セットに後続するテープ・マークは，一連のトレーラー・ラベルの始まりを知らせる．
- トレーラー・ラベル群に後続する最初のテープ・マークは，トレーラー・ラベルの終りを示す．
- データ・セットの最後を示すトレーラー・ラベル（EOF のラベル）に後続するテープ・マークは 2 個あり，その 2 番目のテープ・マークは該当するデータ・セットは，これ以上ないこと，つまり，他のボリュームにもはやまたがらないことを意味する．

これらテープ・マークは，標準ラベルを持つデータ・セットを作成するさいに，データ管理プログラムで自動的に所要個所に書き出される．

なお，非標準のテープ・ラベルは，どんな形式であってもよく，したがって，その処理はすべてユーザーが作成するラベル処理ルーチンに任される．また，ラベルをもたない磁気テープには，データ・セットとテープ・マークしか存在しない．

ラベルが標準であるか，非標準であるかあるいはラベルなしかは，ジョブ制御ステートメントの中の DD ステートメントで指示する．この DD ステートメントのパラメータの1つにLABEL という項目があり，これを使って磁気テープ・データ・セットのラベルに関する情報を与える．標準ラベルだと SL, 標準ラベルとユーザー用ラベルを持つ場合には SUL, 非標準ラベルだと NSL, ラベルなしの場合には NL，ラベルの処理をバイパスしたいときは BLP という指定を LABEL パラメータで行うのである．もし，LABEL の指定を省略すると，当然のことながら，標準ラベルを持つものと解釈される．

LABEL パラメータでは，ラベルの種類以外にデータ・セットの収容位置，パスワードによるデータ・セットの保護，データ・セットの有効期限などの指定もできる．例えば，

```
//OUTDATA DD……， LABEL=
                    (3,SL, PASSWORD, EXPDT=72139),…
```

とすると，これから作成しようとしているデータ・セットは，磁気テープ上の3番目のデータ・セットとして位置させ，そのデータ・セットは標準ラベルを持っており，かつ，パスワードによる保護がなされることを指定している．なお，データ・セットの有効期限は72年の139日目までである．このような形式でラベルに関する情報を与え得るのであるが，その詳細に関しては「ジョブ管理の諸機能とその構造」の章で説明する予定にしている．

標準ラベルに焦点をもどして，その形式と機能に関する説明に入ろう．ヘッダー・ラベルとトレーラー・ラベルは同一形式であるので，磁気テープの標準ラベルの形式は，第4図に示したように4つの形式に集約できる．それぞれの形式について以下で説明してゆこう．

1）ボリューム・ラベルの形式と機能

標準のボリューム・ラベル(VOL1) は，各々のテープ・ボリュームの最初のレコードである．そして，主として，ボリューム通し番号やそのボリュームの所有部門に関する情報を与える（**第4図の1)参照**）．ボリューム・ラベル自体は，磁気テープがコンピュータ部門に納品された時点でユーティリィテイー・

1）テープ・ボリューム・ラベル（80バイト物理レコード）

ボリューム・ラベルの標識（VOL）	ボリューム・ラベル番号(1)	ボリューム通し番号	ボリューム保護コード	ブランク	製造会社用のため予約（ブランク）	ブランク	所有者名と所在地コード	ブランク
3（バイト）	1	6	1	10	10	10	10	29

2）テープ・ヘッダー・ラベル1とトレーラー・ラベル1（80バイト物理レコード）

ラベル標識	ラベル番号(1)	データ・セット名標識名（データ・セット名の下位17バイト）	データ・セット通し番号	ボリューム順序番号	データ・セット順序番号	世代番号	バージョン番号	作成日	有効期限	データ・セット保護標識	ブロック数	システム・コード	ブランク
3	1	17	6	4	4	4	2	6	6	1	6	13	7

ラベル標識：{ヘッダー・ラベルの場合：HDR　トレーラー・ラベルの場合：EOVまたはEOF}

ブロック数：{トレーラー・ラベルのときだけ（ヘッダー・ラベルのときはゼロ）}

3）テープ・ヘッダー・ラベル2とトレーラー・ラベル2（80バイトの物理レコード）

ラベル標識	ラベル番号(2)	レコード形式	レコードの桁数	ブロック・サイズまたはブロック化因数	テープ密度	データ・セット位置	ジョブまたはジョブ・ステップの標識	ブランク
3	1	1	5	5	1	1	17	46

ラベル標識：{ヘッダー・ラベルの場合：HDR　トレーラー・ラベルの場合：EOVまたはEOF}

レコード形式：{固定長の場合：F　可変長の場合：V　無指定の場合：U}

4）ユーザー用ヘッダー・ラベルとトレーラー・ラベル（80バイトの物理レコード，最大8個まで）

ラベル標識	ラベル番号	ユーザーが自由に指定
3	1	76

ラベル番号：ユーザー・ラベルの順番（1から8のいずれか）を示す番号

ラベル標識：{ヘッダー・ラベルの場合：UHL　トレーラー・ラベルの場合：UTL}

第4図　標準テープ・ラベルの形式

プログラムを使って書込んでおくのが普通である．ボリューム番号にしろそのテープの所有部門にしろ，そのボリュームにとっては固定的なものであって，しばしば書き変えられる性格のものではないからである．

入力データ・セットが OPEN されるとデータ管理用ルーチンでは，正しいテープがセットされているかどうかをボリューム・ラベルを利用して確認する．このラベルの中のボリューム通し番号とユーザーが指示した通し番号を突き合わせるのである．ユーザーが指示する通し番号は，DD ステートメントの中で直接指定するか，後述するカタログの機能を通して間接的に行うかのいずれかである．セットされているテープが妥当でないと，そのテープの使用を拒絶し，別の正しいテープをマウントすることを要求するメッセージを出す．正しいボリュームがマウントされ，検査が終ると，データ管理ルーチンは，処理すべきデータ・セットのヘッダー・ラベルのある個所までテープを移動する．このことは，1つのボリュームに複数個のデータ・セットが収容されている場合で2番目以降のデータ・セットを処理するさいには重要な意味をもつ．

出力データ・セットの場合にも，出力テープ・ボリュームの通し番号が指示されているかぎり入力の場合と同様のボリューム・ラベルのチェックが行われる．出力用テープのボリューム番号が指定されていないと，オペレーターに対してスクラッチ・テープをセットさせるためのメッセージが出される．データ管理ルーチンは，そのスクラッチ・テープのボリューム・ラベルからボリューム番号を入手して，所定の制御ブロック（JFCB : Job File Control Block 後述）内に記録しておく．

2）データ・セットのヘッダー・ラベルとトレーラー・ラベル

データ・セットのヘッダー・ラベルとトレーラー・ラベルは，第4図の2)と3)で示したような標準のデータ・セット・ラベル1と2から構成されている．これらラベルがデータ・セットを識別し，データ・セットの属性を定義し，かつ，データ・セットの保護を行うのに使われる．

データ・セット・ラベルは，テープ上にデータ・セットが作られるごとに，データ管理ルーチンによって自動的に作成される．ヘッダー・ラベルとトレー

ラー・ラベルが，つぎの２点を除いては全く同じ形式と内容になっているのは磁気テープの逆読みを可能にするためである．

ヘッダー・ラベルとのちがいは，ラベル標識として HDR の代りに EOF または EOV が使われること，データ・セット上の全ブロックが処理されたかどうかを確認するための手がかりとしてのブロック数が記録されることにある．ヘッダー・ラベルの場合，ブロック数のフィールドは，ゼロにセットされる．

さて，入力データ・セットのラベルの機能について説明しよう．所要のデータ・セットがオープンされたかどうかを確認するために，データ管理ルーチンは HDR1 ラベルに記録されているデータ・セット名とユーザーが DD ステートメントで指定したデータ・セット名を比較する．この比較では，データ・セット名の下位17桁だけが使われる．なお，該当データ・セットが世代別データ群の１つを構成する場合には，世代番号とバージョン番号も比較の対象になることを記憶に留めておいて欲しい．比較の結果が誤りのデータ・セットであることを示した場合には，処理は自動的に中止されて，つぎのジョブに移ってしまう．

HDR1 にあるデータ・セット保護標識は，該当データ・セットが権利のない人々によって使われることを保護するかどうかを示すのに使われる．この個所で機密保護を指定しておくと，データ管理ルーチンは操作員に正しい**合言葉**（password）をキー・インするように要求する．この個所に機密保護のビットを立てるためには，先に示したように DD ステートメントの LABEL パラメーターの中で PASSWORD とすればよい．

ボリューム順序番号は，データ・セットが複数のボリュームにまたがる場合の順番を示し，これによって正しい順序でボリュームを処理しているかどうか確認する．逆に，データ・セット順序番号は，１つのボリュームの中に複数個のデータ・セットが収容されている場合のデータ・セットの順番を示すのに使われる．作成日は，データ・セットが作られた日付が入れられ，有効期限にはこのデータ・セットが何時まで保管される必要があるかを示す．これによって不用意にこのデータ・セットの内容を消してしまわないように保護されるのである．システム・コード欄は，このデータ・セットがいかなる種類のオペレー

ティング・システムのもとで作成されたかを示す．

HDR2 のラベルは，データ管理ルーチンが，データ・セットのある種の性質を決定するのに使用する．このラベルから得られるデータ・セットの属性はレコードの形式，論理レコードの桁数，物理レコード（ブロック・レコード）の長さ，テープ密度などである．これら情報がラベルから得られると JFCB および DCB（後述）に貯えられる．ただし，これら制御ブロックの該当フィールドがゼロの状態にあるときだけ，HDR2 ラベルから必要な属性を示す数値が得られるのである．

なお，ユーザー用ヘッダー・ラベルの処理が不要の場合には，HDR2 のラベルを処理したあとすぐにテープ・マークまで進む．この措置がとられることによって，ユーザー用ラベルはすべてバイパスされ，最初のデータ・レコードの処理可能な状態に入るのである．

データ管理プログラムの中の EOV ルーチンは，データ・セットの終りまたはボリュームの終りの状態を処理する．これら2つの状態は，データ・レコードを処理している過程でテープ・マークをキャッチした時か，処理プログラムの中で FEOV マクロ命令（強制的にボリュームの終りにしてしまう特別なマクロ命令）が実行された時に起る．テープ・マークをキャッチした場合には，そのあとでラベルを読込み，ラベル標識が EOV1 か EOF1 かをチェックする．いずれの標識でもない場合には，誤りであるとみなして処理を中止とする． FEOV マクロ命令の実行結果によるボリュームの終りの状態の場合にはトレーラー・ラベルの検査は行わない．

入力データ・セットの全レコードが読取られたことを確認するために，このルーチンでトレーラー・ラベル1にあるブロック数と処理過程で累積されたそれとを比較する．もし，両者のブロック数が一致しない場合には，ユーザーが指定した特別なルーチンにブランチするか，処理を中止するかのいずれかが選択される．

ユーザー用トレーラー・ラベルがあり，かつ，その処理が要求されている場合にはその処理ルーチンに移ることになる．なお，ラベル標識が EOF であると当該データ・セットのすべてが処理されたとみなすが，逆に EOV であると

トレーラー・ラベルを処理したあと再びデータ管理プログラムの中の OPEN ルーチンに制御権を受渡し，つぎのボリューム上のラベルの処理を再開する．

出力データ・セットの場合には，ボリューム・ラベルがチェックされたあと旧ヘッダー・ラベルが存在するかテープ・マークが存在するかをチェックする．いずれも存在しないときは，本来ヘッダー・ラベルが書き込まれるべき位置に何らかの別のレコードが記録されているものとみなして，そのテープを使用することを中止してしまう．

テープ・マークの存在が確認されるのは，ボリューム・ラベルだけしか記入されていない新しいテープまたは初期化し直されてボリューム・ラベルとテープ・マークをもったテープおよびそのボリュームにいくつかのデータ・セットが書き出されており，そのテープの２番目以降のデータ・セットとして書出される場合である．いずれにしろ，この場合には，HDR1 ラベルが存在する代りにテープ・マークがあるわけだから，それ以上のラベル検査は行わず，すぐに新しいラベルの書出しに入る．

旧 HDR1 ラベルが存在する場合には，まず有効期限のチェックを行う．同一ボリューム上に複数個のデータ・セットが収容されていようとも，最初のしかも，磁気テープのラベルにもたされている HDR1 の有効期限だけを検査する．有効期限を調べたあとで，旧 HDR1 ラベル上のデータ・セット保護標識の有無をチェックする．保護標識がないと，そのテープを出力テープと是認しヘッダー・ラベルの書出しに入る．逆に標識がしるされていると，旧 HDR1 ラベルに示されているデータ・セット名とユーザーが DD ステートメントで指定したそれとを比較する．両者が一致しないとそのテープを出力テープとすることを中止する．一致すると合言葉のキー・インをオペレータに要求する．そのあとで，ラベルの書出しに入るのである．

出力データ・セットのトレーラー・ラベルの書き出しにさいしては，特別な検査はない．EOF か EOV かによってラベル標識の使いわけをするだけである．

3）ユーザー用のヘッダー・ラベルとトレーラー・ラベル

最大8個までのユーザー用ヘッダー・ラベルとトレーラー・ラベルを標準のデータ・セット・ラベルの後に続けることができる．その形式は，**第4図の4)**で示しているように最初4桁を除いては，全くユーザーの選択に任される．したがって，ユーザー・ラベルの形式を整えるとか，内容の検査などはすべてユーザー・ルーチンに任せられる．ただし，データ管理ルーチンは，ユーザー用ラベルの書き出しと読み込みを，ユーザー・プログラムからの依頼にしたがって行う．

いずれにしろ，標準のデータ・セット・ラベルの他にユーザー・ラベルの機能を組んだことは，標準ラベルによる制約を少なくして，ユーザーの自由度と選択権を発揮できる余地を残すことによって，標準ラベルの利用度を高めることを目論んだものである．

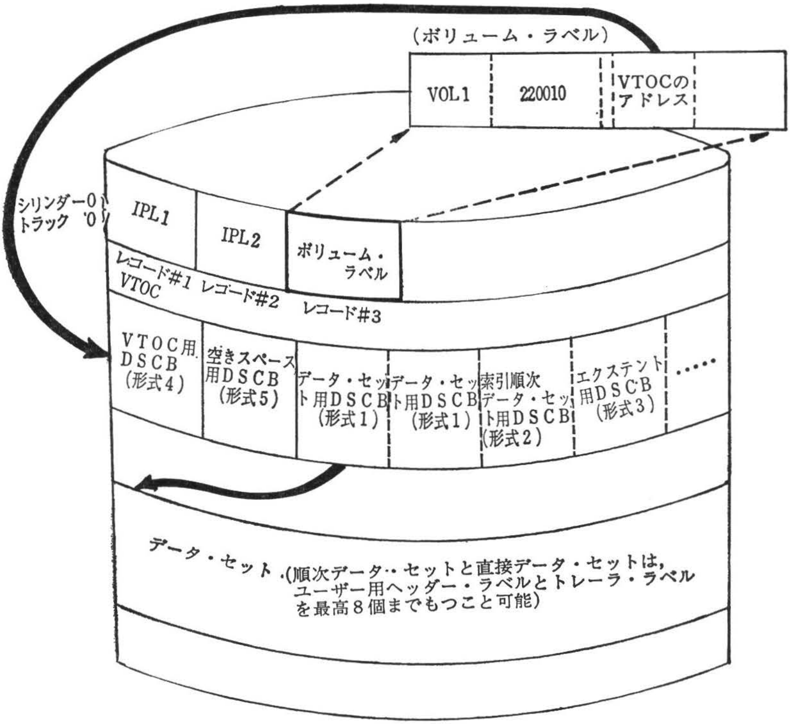
2.　DASD のボリューム・ラベルとデータ・セット・ラベル

DASD のボリューム・ラベルにしろ，データ・セット・ラベルにしろその基本形式と内容およびその目的は，磁気テープの場合と変りはない．ただし，DASD という装置がもっている物理的特性を考慮したデータ管理を行うために磁気テープのラベルには見当たらなかった情報や形式をもったラベルが設計されている．しかし，磁気テープのラベルにもたされていたような機能はほとんど包含している．したがって，それら機能や内容は既知のものとして，本節では特に DASD ラベル特有のものに焦点を合わせて論んじてゆくこととしよう．

直接アクセス・ボリュームは，特定の場所に記憶されたボリューム・ラベルによって識別される．このボリューム・ラベルの形式と内容は，1つの点を除いて磁気テープのボリューム・ラベルと全く同じである．その1つの点とは，データ・セット・ラベルをまとめて収容する記憶域，つまり VTOC の存在場所を示すアドレスが記録されることである．その形式は，**第5図の1)**に示した通りである．「VTOC への指標」というフィールドが，VTOC のアドレスを示すのに用いられる．VTOC が何番目のシリンダー（CC）の何番目のトラック（HH：Hは READ/WRITE HEAD の意味）上の何番目のレコード（R）から始まるというアドレス形式で示される．ボリューム・ラベルには，**ボリュー**

1） DASDボリューム・ラベルの形式

ボリューム・ラベル標識(VOL)	ボリューム・ラベル番号(1)	ボリューム・通し番号	ボリューム・保護コード	VTOCへの指標(CCHHRbbbbb)	製造会社用のため予約(ブランク)	ブランク	所在地コードと所有者名	ブランク
3	1	6	1	10	10	10	10	29



2） VTOC内のDSCBの種類と目的

形式4	形式5	形式1	形式2	形式3	形式0
VTOC自体のエクステント（範囲）の記述	該当ボリューム上の空きのスペースについて26エクステントまで記述	データ・セットの主要な属性とそのデータ・セットの3つまでのエクステントを記述	索引順次データ・セットだけに使われ，形式1で示される属性以外のものを記述	形式1のDSCBで記述しきれないデータ・セットのエクステントの記述	DSCBとして使用されていないVTOC内の140バイトのエリア

第5図　DASD上のボリューム・ラベルとDSCB

ム通し番号のフィールドがあり，ディスク・パックやデータ・セルのような取りはずし可能なボリュームに対して正しいボリュームがセットされているかどうかを確認するのに使われる．また後述するカタログ・システムにおいて，指定されたデータ・セットをつきとめるための手がかりとしても重要な役割りを果すのがボリューム通し番号である．なお，ボリューム・ラベルは，シリンダー番号0でトラック番号0の第3番目のレコードとして記憶される．

VTOC (Volume Table Of Contents の略) は，該当するボリューム上に収容される各々のデータ・セットのデータ・セット・ラベルを一括記録する場所であり，DASD 上の任意の連続した場所に確保される．直接アクセス・ボリューム上に記憶されるデータ・セットのデータ・セット・ラベルでしかも VTOC に収容されるラベルは，特に DSCB (Data Set Control Block) と呼ばれる．したがって， VTOC は，DSCB を一括収容する場所であると言い直すことができる．ボリューム・ラベル，VTOC, DSCB およびデータ・セットの相互関係を図示したのが，**第5図**の中央である．

ボリューム・ラベル内の「VTOC への標識」から VTOC が存在する記憶域の最初のアドレスがつかめる．VTOC の中には，いくつかの形式の DSCB が存在するが，形式1ないし形式3の DSCB を参照することによって該当するデータ・セットの収容されている記憶域（1つの連続した区域とは限らない．いくつかの不連続の記憶域に散在することもしばしばあるが）をつかむことができる． DASD の場合にも，ユーザー用ヘッダー・ラベルとトレーラー・ラベルを持つことができるが，これは VTOC に記憶されるのではなく，本来のデータ・セット記憶域の一部に記録される．磁気テープの場合の形式と全く同じでその用途や取扱いも磁気テープのそれと同じである．なお，DASD 上のラベルには，磁気テープの場合のように非標準ラベルであるとかラベルなしのデータ・セットは許されない．必ず，ボリューム・ラベルと所定の DSCB を持ったデータ・セットである．

VTOC に収容されるデータ・セット・ラベル，つまり， DSCB は，140 バイトの長さを持ち（キー部が44バイト，データ部が96バイト），データ管理プログラムでデータ・セットのアクセスを制御するために必要となる情報をもって

いるが，**第5図の2)**で示したように５種類の形式のものがある．形式１は，通常のデータ・セットには必須のものであり，磁気テープの HDR1 と HDR2 の内容のほとんどが収容される．それ以外に，該当データ・セットが記録される記憶域（**エクステント**という）が３つまで記述される．もし，該当データ・セットがそのボリューム上に４つ以上のエクステントを持つ場合には，形式３で４番目以降のエクステントが定義される．索引順次編成のデータ・セットだけは，形式１以外に必ず形式２の DSCB が作られる．この形式では，形式１で定義しきれない索引順次編成特有の諸属性について定義される．形式４の DSCB には，VTOC 自体（VTOC そのものも１つのデータ・セットとみなされるのであるが）の記憶域に関する情報が貯えられる．さらに，形式５には，このボリューム上に散在する未使用ないしは他のデータ・セットに使用可能な空きのスペースを管理するための情報が貯えられる．また形式０と呼ばれる

1）形式１の主要項目

データ・セット名 (44バイト)	作成日	有効期限	第1エクステント		第2エクステント		第3エクステント		形式3 形式2 の位置
	YYDDD	YYDDD	下限	上限	下限	上限	下限	上限	CCHHR
			CCHH	CCHH	CCHH	CCHH	CCHH	CCHH	または 00000

2）形式３の項目

第1追加エクステント		第2追加エクステント		第3追加エクステント		第4追加エクステント		第12追加エクステント		第13追加エクステント	
下限	上限	下限	上限	下限	上限	下限	上限	下限	上限	下限	上限
CCHH	CCHH	CCHH	CCHH	CCHH	CCHH	CCHH	CCHH	CCHH	·CCHH	CCHH	CCHH

3）形式５の項目

空きのエクステント−1			空きのエクステント−2				空きのエクステント−26			後続の形式5の位置
最初のトラックアドレス	シリンダー数	トラック数	最初のトラックアドレス	シリンダー数	トラック数		最初のトラックアドレス	シリンダー数	トラック数	CCHHR または 00000

4）形式４の主要項目

VTOC内にある最後の有効DSCB（ただし，形式1）の位置	VTOC内の未使用のDSCBの数	

第6図　主要なＤＳＣＢの形式

DSCB は，正規の DSCB ではなく，まだ DSCB として使用されていない 140 バイトの空きの DSCB 用のエリアであり，それを便宜的に形式0と称しているに過ぎない．

以下に主要な DSCB について詳述することにしよう．主要な DSCB に関してその形式を**第6図**にまとめてある．

1） 形式1の DSCB

この形式は，VTOC 以外のすべてのデータ・セットを識別するための情報が入っている．したがって，該当ボリュームに収容されている各々のデータ・セットに関して必ず1個の形式1の DSCB が用意される．磁気テープと異なり DASD ボリュームの場合には，通常1つのボリュームに数個から数十個のデータ・セットが収容される．それぞれのデータ・セットに対して1個づつ形式1の DSCB が作成されるのである．

第6図の1)で示しているように，データ・セット名のフィールドは44バイトあり，完全なデータ・セット名で記録できるようになっている．この使用目的は，磁気テープのそれと同じである．その他にデータ・セットの作成日，有効期限，世代別データ群の場合の世代番号とバージョン番号，データ・セットの保護標識，レコードやブロックの長さ，データ・セットの編成法やレコードの形式などの情報が貯えられており，磁気テープの場合と同様の機能を果す．さらに，3つまでの連続したデータ・セットの収容エリア，つまりエクステントを示すフィールドがある．このフィールドでは，そのデータ・セットに割り当てられた記憶域を，どのシリンダー（CC）のどのトラック（HH）からどのシリンダーのどのトラックまでという下限と上限で示す．この形式の最後には，連鎖すべき他の DSCB のありかを示すフィールドがある．索引順次編成のデータ・セットだとそのデータ・セットに関連する形式2の DSCB が記録されている場所のアドレスを示す．一方，索引順次以外のデータ・セットでは該当ボリューム上に点在するエクステントが4つ以上になる場合は，4番目以降のエクステントを記述する形式3の DSCB がある場所のアドレスを示す．エクステントが3つ以下の場合には，このフィールドは自動的にゼロとされる．

2) 形式3の DSCB

この形式の DSCB は，形式1の DSCB で定義しきれなかった追加のエクステントの定義をする．第6図の2)でその形式を図示しているように最高13個までのエクステントが定義できる．形式1では，3個までのエクステントが記述できるので，都合16個までのエクステントが1つのデータ・セットに許されることになる．ただし，これは，1つのボリューム上でのエクステントの制約であって，あるデータ・セットがいくつかのボリュームにまたがる場合には，それぞれのボリュームについて16までのエクステントが許容範囲ということになる．形式3の DSCB は，索引順次編成のデータ・セットでは形式2の DSCB から連鎖され，その他の編成のデータ・セットでは形式1のデータ・セットから連鎖される．

3) 形式5の DSCB

この形式の DSCB は，データ管理プログラムが直接アクセス・ボリューム上のスペースを管理してゆくうえでの貴重な情報が貯えられる．つまり，データ・セットに割り当てることのできるすべての空きスペースの情報がこの DSCB に貯えられるのである．形式5の DSCB 1個で最高26個までの利用可能なエクステント（新たなデータ・セットに割当て可能な記憶域）の記述ができる．26個を越える空きのエクステントが生じた場合には，更に別の形式5の DSCB を作り，前の DSCB から連鎖することができる（第6図の3) 参照)．各々の空きのエクステントの定義は，空きのエクステントがどこから始まり（CCHH)，何シリンダーと何トラック分のスペースであるという形式でなされる．

空きのエクステントは，単にまだ未使用であるスペースによって生じるだけでなく，あるデータ・セットが削除されることによっても生じる．新しいデータ・セットに対するスペースの割り当てと古いデータ・セットの削除が何回かくり返されると，とても DASD 上のスペースを有効に割り当ててゆくことはむずかしい問題となってしまうが，スペースの管理を完全にデータ管理プログラムに委ねることによって人的負担を軽減するだけでなく，より効率的なスペ

ース管理が実現できるのである．このための重要な情報源が，この DSCB である．なお， DASD 上のスペース管理の詳細は，「直接アクセス・ボリュームのスペース割当て機能」の項で言及する．

形式5の最初の DSCB は， VTOC 内での第2番目のブロックとして位置づけられる（第5図参照）．第1番目のブロックは， VTOC 自体の DSCB，つまり形式4である．

4）形式4の DSCB

VTOC は， そのボリューム上に収容されるデータ・セットの記述を行うデータ（一連の DSCB）から構成されるがゆえに，やはり，一種のデータ・セットとみなされる．それゆえに， VTOC 自体も，自己のエクステントを記述する DSCB を持つ必要がある．この DSCB が形式4である．形式4の DSCB で記述される主要な内容は，第6図の4）で示しているように，目下の VTOC 内での利用状況を示す． VTOC 内のどこまでが使用ずみであり，あと何個分の DSCB を記入できる余裕が残されているかを示しているのである．

以上，直接アクセス・ボリュームのボリューム・ラベルとデータ・セット・ラベルである DSCB について説明してきたが，これらラベルの作成，管理およびその利用はすべてデータ管理プログラムを通じて行われるわけである．決して，プログラマーなりオペレータなりが直接かかわりをもつものではなく，プログラマーなりオペレータの肩代りをデータ管理プログラムが行うために必要な情報が集約化されているのがラベルなのである．なお，ボリューム・ラベルの作成と VTOC のエリアを準備するためにはユーティリィティ・プログラムを使用する．ボリューム・ラベルは固有の場所に作り上げられるが，VTOC の場所とその大きさは任意に指定することができる．

五. カタログ・システムの機能と構造

1. カタログ・システムの基本的機能範囲

コンピュータ利用の規模が大型化へ発展し，コンピュータ自体は新世代へと移行してゆくと，そのシステムの内部処理速度は高度化する一方であり，かつ補助記憶装置は多様化し，その記憶能力も大容量化してゆく．その結果，1つのシステムで取扱う仕事はこれまでになく増加し，管理すべきデータ・ファイルの種類はぼう大な数に昇り，データ・ファイルを管理する仕事は相当な負担になってくる．殊に，直接アクセス記憶装置（DASD）が主要な補助記憶装置の地位を占めるようになってくると，それがもっている本来的能力をフルに活用する方向が目指されるわけで，データ・ファイルの管理という問題も一段と困難な課題となってきた．

急増するデータ・ファイルの大半が，補助記憶装置，つまり，磁気テープやDASD に貯えられるのが最近の傾向である．しかるに，旧来のデータ管理の方法だと，オペレーションを担当するスタッフが，それぞれのデータ・セットを貯えるボリュームとスペースを手作業で割り当てる必要がある．磁気テープの場合には，どのボリュームをそのデータ・セットに割り当てるかを決定するだけですむが， DASD ボリュームだと，さらに，そのボリュームのどのスペースを割り当てるかをも決定する必要がある．いずれにしても，オペレーション・スタッフは，データを記憶するためのスペース割り当ての記録を保持しそれに従ってデータ管理を組織的に行う必要があった．

一方，プログラマーにとってみれば，自分が作成するプログラムで利用するデータ・セットがどのボリュームに貯えられるのかをあらかじめ知る必要があった．特に DASD の場合だと，単にどのボリュームに貯えられるかだけでなく，そのボリューム上のどのアドレスからどのアドレスまでの範囲が該当データ・セットに割り当てられるかを知らないことにはプログラムが作成できなかった．プログラムの作成とオペレーションを同一要員でカバーするのならまだ

しも，プログラマーの職能とオペレータの職能が分化してきた現在，データ・セットの管理手順のもとで，プログラマーの作業を制約することは必ずしも容易なことではない．

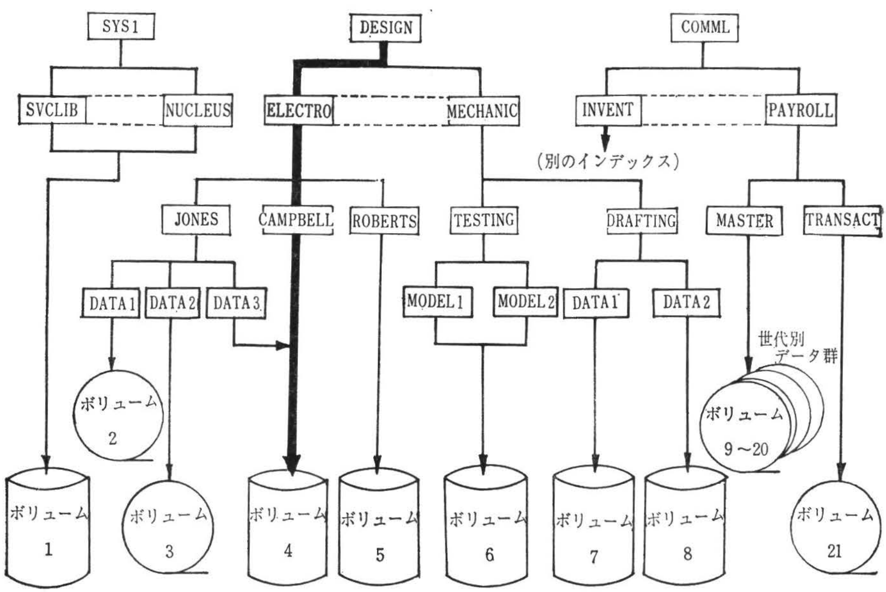
しかし，次々と新しいデータ・セットが作成され，古いデータ・セットは廃棄されるという事態にあって，ますます統一的なデータ・セットの管理の必要性が増すが，一方その管理は極めて困難な問題となってきた．単にデータ・セットの急増と代謝にいかに対処するかという問題だけでなく，データ管理過程への人的介入の増加によって引き起されるエラーへの対処という問題も投げかけられてきた． DASD 上のスペースの重複割当てのような事態が発生し，データ・セットが破壊されるような危険が生じてくると，プログラマーはこれまでのデータ管理の手順に従い得なくなり，自分が関係する仕事で使うデータ・セットはそれ専用の独自のボリュームを持つ傾向がでてきてしまう．当然，別の仕事と同一データ・セットや同一ボリュームを共有するということが減少してゆき，ボリュームの効率的使用が望めなくなると同時に，統一的データ・セットの管理も望めない状況となってきた．

その一方では，多くのコンピュータ・ユーザーが DASD の効率的利用という問題に直面したことも事実である．1個のボリュームの記憶容量が数千万文字，数億文字という具合いに記憶能力が高まってくると，1個のボリュームに数個から数十個のデータ・セットを共存させる必要がでてくる．ところが，新旧データ・セットの代謝が行われるとスペースの割り当てがうまくゆかなくなり，随所に遊休のスペースが生じてしまう．その結果，本来の記憶能力の数分の1しか活用されない状態になってしまう．そこで，ボリューム上のデータ・セットを再編成し，統合化しようという試みがなされるものの，時間と人手を必要とすることから，かならずしもうまくゆかないのが実情であった．

このような過去のデータ・セット管理上の諸問題を考慮に入れて設計されたのが，データ・セットのカタログ・システムであり，データ管理の主要な機能の1つとしてオペレーティング・システムに組込まれている．このシステムを利用することによって，データ・セットを使用目的に応じて分類し，記憶させデータ・セット名を与えるだけでそのデータ・セットをつきとめることができ

る．このシステムは，多くの点で図書館における図書検索システムに似かよっている．この場合，図書館員の役割りを演ずるのが制御プログラムのうちのデータ管理プログラム（特に，**カタログ管理ルーチン**と呼ばれる部分）であり，検索用図書目録の役目を果すのが，**カタログ**である．つまり，カタログ管理ルーチンとカタログの２つの要素でカタログ・システムが構成されるのである．カタログには，データ・セットが存在するボリュームを識別するのに必要な情報が貯えられており，カタログ管理ルーチンは，カタログそのものを作り上げたり，カタログを通じて必要なデータ・セットをつきとめたり，必要に応じてカタログを更新する機能を果す．したがって，あらかじめカタログにデータ・セットの登録がなされているなら，データ・セット名が与えられるだけで，そのデータ・セットが保持されているボリュームの識別を行い，該当データ・セットのありかをつきとめてくれるのである．

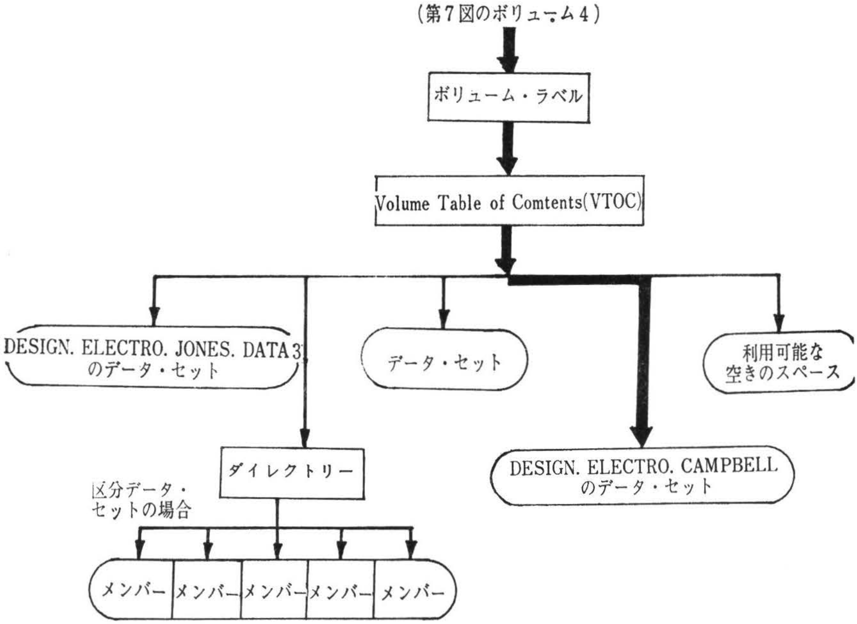
特定データ・セットをつきとめるために使われるカタログは，階層別のインデックスから構成され DASD 上に収容される．カタログの基本的構造と特定



第７図　カタログの樹木式構造例と特定ボリュームの探索の仕方

ボリュームの検索のステップを図示したのが**第7図**である．この図から推測されるように，カタログは単にカタログ管理ルーチンがデータ・セットをつきとめる役割りを果すだけでなく，データ・セットを一定の体系に分類管理するのにも都合よく工夫されている．この一定の分類体系が確立できるようにしたのが，階層別インデックスの思想である．別の表現にしたがえば，樹木式構造ということになる．先にもふれたように補助記憶装置に収容されるデータ・セットは，人手での管理では間に合わないほどの数にのぼっている．これらデータ・セットに同一名を与えることなく体系的な管理を行ってゆくためには，図書館における図書分類法と同じような分類手法を採用するのが一番ふさわしい．図書分類では分類コードを採用しているが，データ管理においては，分類コードの代りに8桁までの英数字名を使って，データ・セットを識別する方法をとる．たとえば，DESIGN. ELECTRO. CAMPBELL というデータ・セット名を例にこれを考えてみよう．この場合，CAMPBELL は，ELECTRO という分類項目のもとで分類される基本項目であり，ELECTRO は DESIGN という大分類項目のもとでの分類項目である．それぞれの分類項目名が，カタログ内のインデックスに対応するわけである．したがって，カタログ管理ルーチンでは DESIGN. ELECTRO. CAMPBELL というデータ・セット名が与えられると，まず大分類項目である DESIGN という名前をつかって，カタログ内の対応するインデックスを検索する．カタログ内で DESIGN というインデックスをつきとめると，つぎは ELECTRO という項目名で対応するインデックスを検索する．ELECTRO が見つかれば，CAMPBELL という項目名で対応するインデックスを検索し，ついにはこのデータ・セットが収容されているボリュームの通し番号をつきとめる（第7図では，ボリューム4）．該当ボリューム番号をもったボリュームがセットされていないときには，しかるべき装置に該当ボリュームを取付けるようにオペレータに指示を行えばよい．

制御プログラムのカタログ管理ルーチンで，データ・セットが収容されているボリュームを識別すると，つぎはボリューム内のどの場所にそのデータ・セットが存在するかをつきとめる必要がある．特に直接アクセス・ボリュームの場合だと，ボリューム・ラベルを通じて VTOC の記憶域のアドレスをつか



第8図 特定ボリューム内でのデータ・セットの探索順序

む. ご存知の通り VTOC には, そのボリューム内に収容されている各データ・セットの名前やそのエクステントなどが DSCB として集約されているので, これを手がかりにしてデータ・セットを見出すことができよう. この様子を示したのが**第8図**である. VTOC には, そのボリューム内での空きのスペースを管理する DSCB（形式5）があるので, 制御プログラムはこの DSCB を使って新しいデータ・セットに必要となるスペースを自動的に見つけ出し, そのデータ・セットに割り当てることができる. したがって, 旧来のデータ管理方式でプログラマーやオペレータに負担がかかったスペースの管理からも解放されるのである. なお, このスペースの管理機能は, カタログ管理ルーチンの働きではなく, スペース管理ルーチンの機能であるので詳細はそちらでふれる.

なお, テープ・ボリュームには VTOC はないが, 1つのテープ・ボリュームに2つ以上のデータ・セットが収容される場合には, データ・セットが作成される時点でボリューム上のデータ・セットの順番を示す番号（データ・セッ

ト順序番号）が割当てられるので，これを使って該当データ・セットが位置する個所まで自動的にテープを前進させてくれる．

さて，カタログ・システムのもとで管理されるデータ・セットであるが，データの形式によって何らの差別もされない．ジョブ制御ステートメントの集まりからなるデータ・セットであろうが，原始プログラムや目的プログラムの集まりから構成されるデータ・セットであろうが，あるいは，処理プログラムでプロセスされるデータ・ブロックから成るデータ・セットであろうが全く同じ扱いをうける．したがって，オペレーティング・システムを構成し，それによって利用されるすべてのデータ・セットがカタログ・システムに登録できる．カタログ・システムは，すべてのデータ・セットに解放されたシステムとして設計されているのである．一たんオペレーティング・システムが生成されるとカタログ・システムのために特別に作成されたユーティリィティー・プログラムを使って自由にカタログを拡充してゆくことができるように工夫されている．たとえば，いくつかのレベルで構成されるインデックスからなるカタログを作り上げたり，データ・セットの増加に応じてカタログ自体を拡大したり，新たな分類体系にカタログ・インデックスを再編したりが自由に行える．また，カタログを構成する最上位のレベルのインデックス（第7図では，SYS1, DESIGN および COMML） だけは，必ずシステム・レジデンス・ボリュームに収容する必要があるが，それ以下のレベルのインデックスは，別のボリュームに分散収容することもできる．たとえば，第7図で DESIGN というインデックスの支配下にあるインデックス部分だけを設計部門だけの専門のディスク・パックに分散収容できるのである．

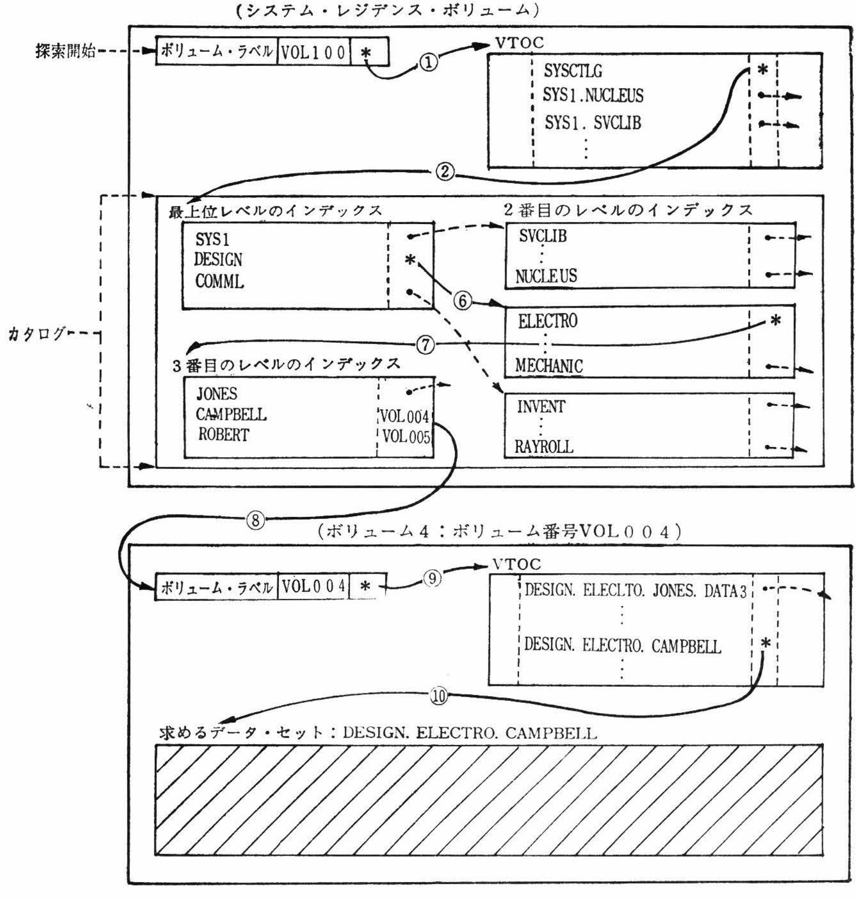
カタログ・システムには，単一データ・セットのカタログだけでなく，世代別データ群を登録管理することもできるし，区分データ・セットの登録管理もできる．区分データ・セットは以前にもふれたように主としてプログラムを記憶するためのデータ・セットの編成法であり，一般にプログラム・ライブラリーと呼ばれるものである．第8図の一部で示しているように，このデータ・セットはダイレクトリーと呼ばれる独自のインデックスをもっている．このインデックスは，区分データ・セットを構成するメンバーと呼ばれるデータ・ブロ

ックをつきとめるために使われる．メンバーは，1つ1つのプログラムに相当し，ダイレクトリーに収容されているインデックスがプログラム名に当てるものである．特にシステム・データ・セットと呼ばれる一連のプログラム・ライブラリーはこの編成をとっており，必ずカタログに登録される．

以下の項目で，カタログ・システムの機能的構造，世代別データ群の構造とカタログ，システム・データ・セットの構成とカタログを中心に説明してゆこう．

2. カタログ・システムの機能的構造

カタログ・システムのカタログを収容するデータ・セットは，DASD ボリュームに記憶する必要があることはすでに述べた通りである．このカタログを収容するデータ・セットはシステム・データ・セットの一種であり，そのデータ・セット名は SYSCTLG と名付けられている．カタログに登録するすべてのデータ・セットのインデックスがこの SYSCTLG に記録され，カタログ管理ルーチンによって管理される．SYSCTLG データ・セットは，システム・レジデンス・ボリュームに記憶される性質のデータ・セットである．当然のことながら，このカタログを収容するためのデータ・セットのデータ・セット・ラベル，つまり，DSCB はシステム・レジデンス・ボリュームの VTOC に貯えられる．それゆえ，システム・レジデンス・ボリュームの VTOC を，ボリューム・ラベルを手がかりにつきとめることによって，SYSCTLG の記憶位置を知ることができる．**第9図**でその様子を示している．ただし，この例はカタログの全体がシステム・レジデンス・ボリューム上の SYSCTLG に収められる場合のものである．言うまでもなく，SYSCTLG の DCSB では，カタログが記憶されている記憶域がどこに存在するかをエクステントで指示してくれる(第9図の②)．カタログ内には，まず最上位のインデックスが示される．第9図，第10図のインデックスはいずれも，第7図のカタログの分類項目にしたがっている．最上位のインデックスは，特に**ボリューム・インデックス**と呼ばれ，第7図の分類にしたがうと SYS1, DESIGN および COMML がボリューム・インデックス内に収容される．ボリューム・インデックスの各々の記入



第9図　カタログ全体の探索
(カタログ全体がシステム・レジデンス・ボリュームにある場合)

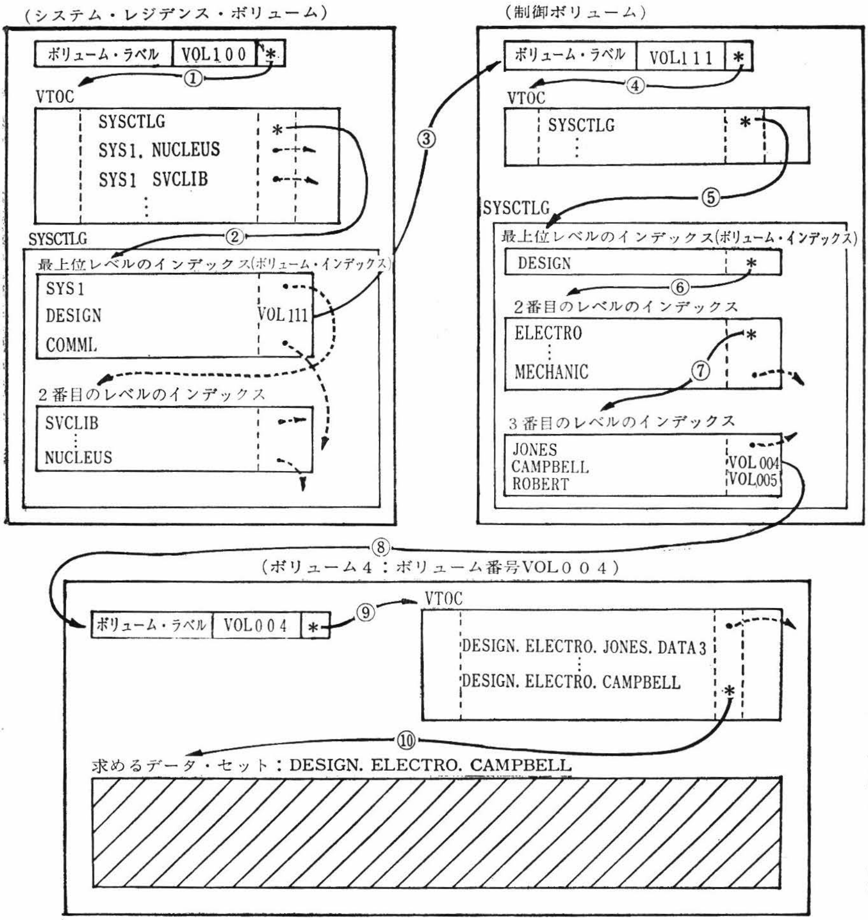
項目には，次のインデックスがカタログ内のどこに貯えられているかのポインターを持っている（第9図の⑥). DESIGN. ELECTRO. CAMPBELL というデータ・セット名の場合だと，2番目のインデックスは，DESIGN のインデックスで指示され，その中には，ELECTRO とか MECHANIC の分類項目がある．

ELECTRO が当面のデータ・セット名の分類名であるので ELECTRO のイ

ンデックスがどこに存在するかが2番目のインデックスで確認される．ELECTRO の内部分類項目を示すインデックスは，3番目のインデックスである．3番目のインデックス内の分類項目 CAMPBELL をつきとめると求めるデータ・セットの収容されているボリューム番号が判別する(第9図の⑧)．このように最下位の分類名に対応するインデックスの記入項目では，該当するデータ・セットが収容されているボリューム番号が貯えられている．ボリューム番号以外に該当ボリュームの装置の種類，そのデータ・セットを収容するのに使われているボリュームの個数および磁気テープの場合のデータ・セット順序番号なども貯えられている．求めるデータ・セットのボリューム番号がつかめると，あとは目的とするボリュームが入出力装置にセットされているかどうか確認し，セットされておればボリューム・ラベルを通して，そのボリュームの VTOC のアドレスをつかみ，VTOC 内の該当データ・セット名をもつ DSCB を見出すことによって，該当するデータ・セットの記憶場所を知ることができるのである（第9図の⑨，⑩).

カタログの最上位のレベルのインデックス，つまり，ボリューム・インデックスは，システム・レジデンス・ボリューム上の SYSCTLG に収容される必要があった．第9図で示したようにカタログ全体をシステム・レジデンス・ボリュームに置くこともできるが，第10図の例のようにカタログの一部を別のボリュームに分割収容することもできる．このようにカタログのある一部を収容しているボリュームのことを，特に**制御ボリューム**と呼ぶ．制御ボリュームを複数個作ることによって，機能的に関連するデータ・セットだけを別々に分離してカタログできるようになる．

さて，**第10図**を参照しながら，カタログの一部を分割収容した場合のケースを考えてみよう．分割収容したカタログ・データ・セットもその名前は依然として SYSCTLG である．第10図では，DESIGN という分類名の支配下にあるインデックスだけを，ボリューム番号VOL111 に収めている．ボリューム・インデックスだけは何時でもシステム・レジデンス・ボリュームに置かれているので，所要のデータ・セットの検索の開始は，このボリュームから始まる（第10図の①と②)．カタログの一部が他のボリュームに収容されている場合のボリ



第10図　カタログの探索（カタログの一部が別の制御ボリュームにある場合）

ューム・インデックスの記入項目のポインターの個所には，そのカタログ部分が収められている制御ボリュームの番号が入っている．このボリューム番号を手がかりに制御ボリュームをつきとめる（第10図の③）．なお，システム・レジデンス・ボリュームのカタログと他のボリュームに入っているカタログを有機的に結ぶつける（具体的には，システム・レジデンス・ボリュームのボリューム・インデックスの記入項目の一部に制御ボリュームの番号を挿入する）仕事は、後述するシステム・ユーティリィティー・プログラムで達成される．

制御ボリュームがつきとめられると，ボリューム・ラベルから VTOC のアドレスを得て，VTOC 内の DSCB を探索する．その結果，SYSCTLG の DSCB からカタログが収容されている場所がわかる（第10図の④と⑤）．このカタログの最上位のインデックスは，やはりボリューム・インデックスである．あとは，第９図の場合と同じようなステップをたどって，求めるデータ・セットをつきとめることができる（第10図の⑥から⑩）．

このように，カタログを分割収容するといくつかの利点が得られよう．まず第１に，複数台のコンピュータ・システムを利用しているユーザーにおいて，あるシステムから別のシステムに制御ボリュームを移動することによって，同一カタログをシステム間で共有できよう．このことは，処理すべき仕事をあるシステムに固定することを避けて，弾力性に富んだシステム利用を実現可能にする．第２に，使用頻度の低いデータ・セットのインデックスを別の制御ボリュームにカタログしておくことによって，システム・レジデンス・ボリューム自体のスペースの有効利用が計れる．また，有機的に関連したカタログを一括して分割収容できることから，体系的なデータ・セットの保護管理が達成できるなどの利点がある．

以上みてきたようなカタログの構造を作り上げ，カタログを通してしかるべきデータ・セットを探し当て，あるいはカタログ自体を更新する機能を果すのが，言うまでもなくデータ管理プログラムのうちのカタログ管理ルーチンである．カタログ管理ルーチンの主要な機能は，LOCATE 機能，INDEX 機能，CATALOG 機能の３つに大別される．**LOCATE 機能**とは，与えられたデータ・セット名をもとにカタログ内のインデックスを検索し，カタログされたデータ・セットの存在するボリュームをつきとめる機能であり，第９図，第10図をもとに説明したような手順でその機能を果す．**INDEX 機能**とは，カタログすべきデータ・セットのインデックス構造を作り上げたり，逆にすでにカタログに収容されているインデックスのうち不要なものを削除したりする機能である．３番目の**CATALOG 機能**は，データ・セットの基本項目名，つまり最下位の修飾名とボリューム番号をカタログ内に登録したり，逆に削除したりする機能である．

カタログ管理ルーチンは，システム・ユーティリィティー・プログラムを通して利用されるか，ジョブ制御ステートメントのうちのDDステートメントを通して要求されることにもとづいて機能する．INDEX 機能は，カタログすべきインデックスを作成するためにユーティリィティー・プログラムで利用される．一方 LOCATE 機能は，入力データ・セットをカタログを通して見つけ出すためにジョブ管理プログラムによって利用される．CATALOG 機能は，ユーティリィティー・プログラムとジョブ管理プログラムの双方で，新しく作成されたデータ・セットをカタログするのに使われる．また特別なケースとしては，LOCATE マクロ命令，INDEX マクロ命令あるいは CATALOG マクロ命令を使って，その機能を利用することも可能である．

すべてのインデックス・レベルを包含したカタログ構造を作り上げたり，そのカタログ構造に修正や追加を行うのが，システム・ユーティリィティー・プログラムの一種である IEHPROGM である．このプログラムが行ってくれる具体的機能としては，

- インデックスの作成および削除．
- 制御ボリュームの連結または分離．
- 世代別データ群のインデックスの作成・保守．
- データ・セットのカタログまたはカタログからの削除．
- データ・セットあるいはメンバー名の変更．

などである．

一方，DD ステートメントを通して，カタログ管理ルーチンの機能を利用する例は，**第11図**に示した．新しく作成したデータ・セットをカタログに登録するには，DISP=(NEW, CATLG) とする．ただし，これによってカタログに登録できるのは，すでに該当するインデックス構造がユーティリィティー・プログラムを使って作成されていることを条件とする．たとえば，データ・セット名 DESIGN. ELECTRO. CAMPBELL でカタログするのには，システム・レジデンス・ボリュームのボリューム・インデックスが DESIGN インデックスの記入項目を持っている必要があるし，DESIGN インデックスは，ELECTRO インデックスのポインターを持っていることが前提になる．この場合，

1）データ・セットのカタログの例

```
//OUTPUT DD DSNAME=DESIGN. ELECTRO. CAMPBELL, DISP=
//            (NEW, CATLG), UNIT=2311, VOLUME=SER=VOLOOA,
//            SPACE=(CYL, 100)
```

2）カタログされたデータ・セットの利用

```
//CAMPDATA DD DSNAME=DESIGN. ELECTRO. CAMPBELL, DISP=OLI
//              DISP=OLD
```

3）カタログからの削除

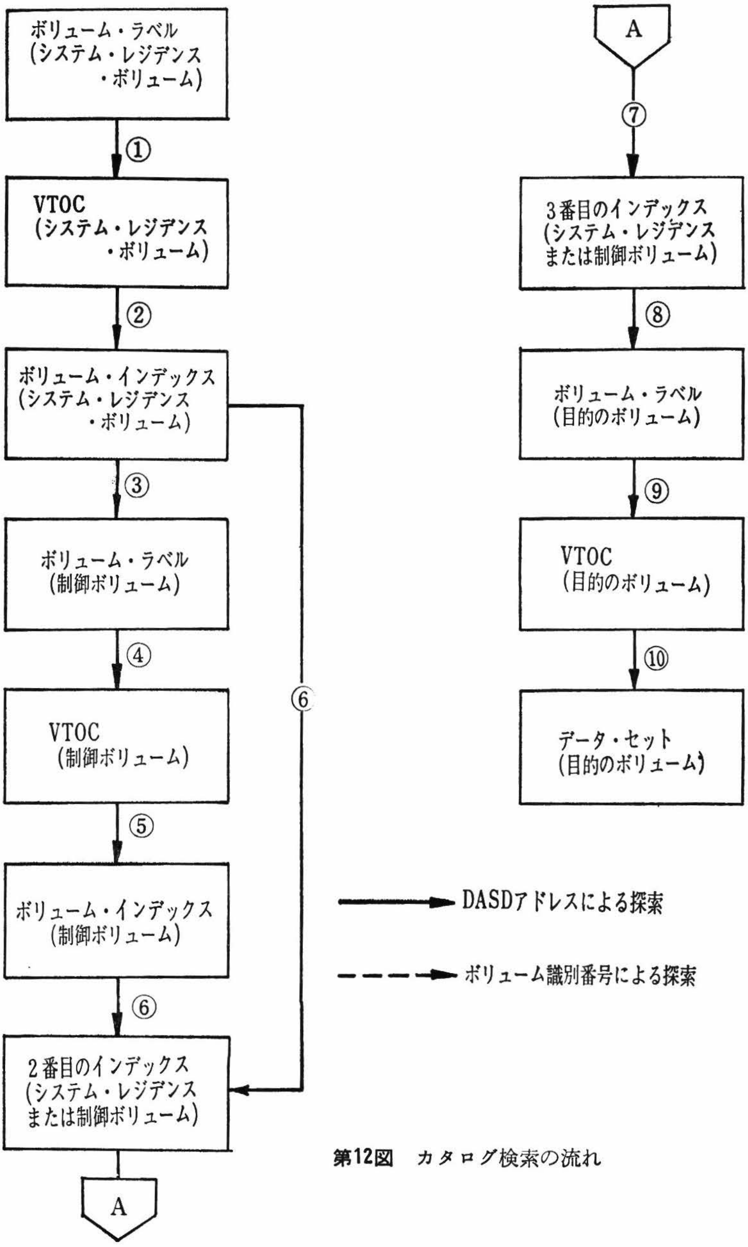
```
//OLDDATA DD DSNAME=DESIGN. ELECTRO. CAMPBELL,
//              DISP=(OLD, UNCATLG)
```

第11図 DD ステートメントによるカタログの取扱い例

DESIGN. ELECTRO. CAMPBELL がカタログされると，ELECTRO インデックスに CAMPBELL が登録され，このデータ・セットが収容されているボリューム番号が記入されるのである．

逆に，カタログされたデータ・セットをカタログから取除くには DISP=(OLD, UNCATLG) とする．そうすると ELECTRO インデックスから，CAMPBELL とそのボリューム番号が削除されてしまうのである．

カタログされたデータ・セットを入力データ・セットとして利用するには，第11図の2）で示しているように，DDステートメントで，データ・セット名(DSNAME=）と DISP を与えるだけで事足りる．装置の種類だとか，ボリューム番号などを最早与える必要はないのである．ジョブ管理プログラムのジョブ・スケジューラーがこのステートメントを解読したあとで，カタログ管理ルーチンの LOCATE 機能を利用して，しかるべきデータ・セットの収容されているボリュームを探し当ててくれるのである．これまで説明してきたカタログ検索，つまり LOCATE の機能を一般化して図示すると第12図のようになる．



第12図　カタログ検索の流れ

以上のようなカタログ・システムの機能構造をとることによって，プログラマーなりオペレーターなりはデータ・セットの管理というわずらわしい仕事の大半から解放されると同時に，コンピュータによる自動管理ということから誤ったデータ・セットの利用や有効データ・セットの破壊といった事態を最少限にくいとめることができるようになる．

3. 世代別データ群の構造とカタログ

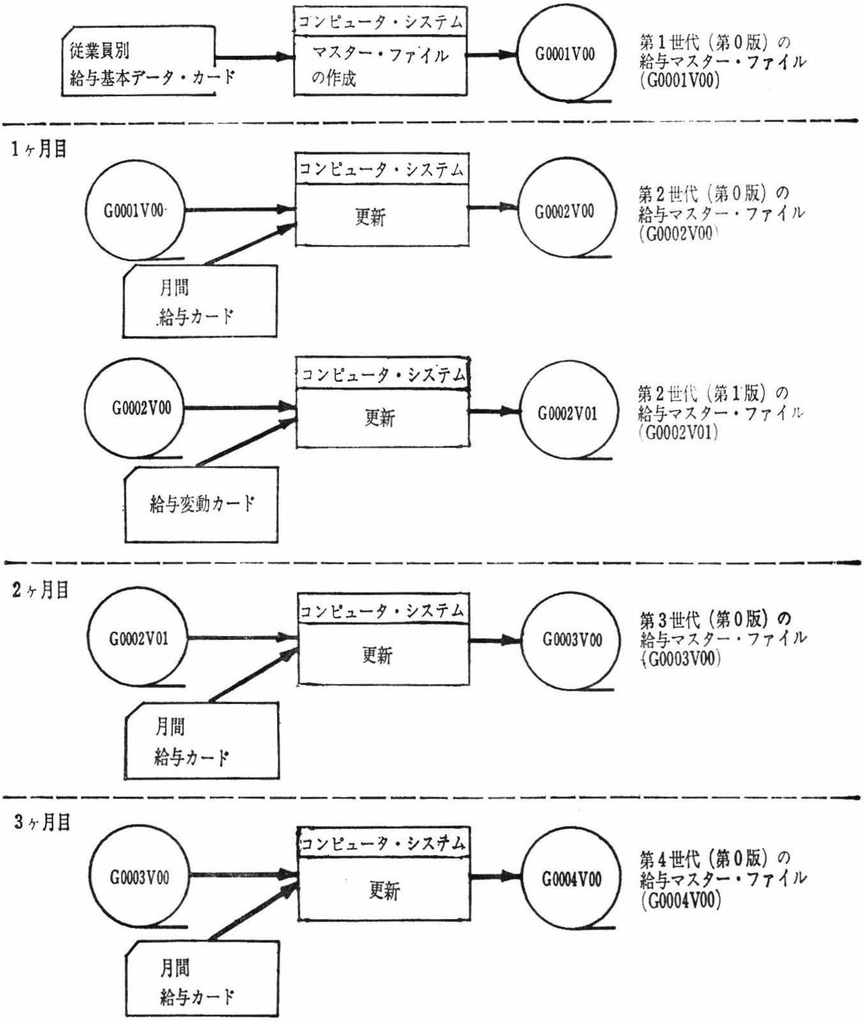
世代別データ群（generation date group）とは，作成時間的にあるいは機能的に相互に関連性をもったカタログされたデータ・セットの集まりのことである．世代別データ群を構成する１つ１つのデータ・セットのことを，**世代別データ・セット**あるいは単に**世代**といった呼び方をする．

カタログ・システムを使うことによって，関連データの連続する世代を分類することができる．これら世代別データ・セットには，全く同一の名前を与えて，最新の世代を基準にした相対的番号でそれぞれのデータ・セットを見分けることができる．そして，カタログ・システムが，自動的に世代番号を更新・保持してくれるのである．

さて，世代別データ群がどのように構成され，カタログ・システムでどのような取扱いをうけ，どのような利用が行われるかを具体例にもとづいて説明してゆこう．

そのために，第７図を再度参照していただきたい．ここには，COMML. PAYROLL. MASTER と名ずけられた世代別データ群が示されている．このデータ群には12個のテープ・ボリュームが割り当てられていることから，12世代のデータ・セット，つまり１年間のデータ・セットの世代別データ群であることが示唆されている．この例では，世代別データ群のために磁気テープ・ボリュームが使われているが，もちろん DASD ボリュームであっても差支えない．

給与マスター・ファイルは，毎月定期的に更新されるのが普通である．更新作業の主要入力は，前月の出力データ・セットである．**第13図**で，オリジナル・マスター・ファイルの作成から３ケ月にわたる給与マスター・ファイルの更



第13図　世代別データ群の例

新の様子を示している．1ケ月目で示しているように，定期的更新以外に特別な変動の理由でその月のマスター・ファイルに手直しを施す必要性が生じることもある．月々の入力に使われたマスター・データ・セットは，後日のために保管するのが普通である．税法上の理由，年末調整のため，あるいは企業のポリシイーとして，長期間のデータ・セットを保管する場合が多い．もし，1ケ年のマスター・データ・セットを保管するとしよう．少なくとも12世代のデータ・セットが保管されることになろう．もし，これまで説明してきたカタログの方法を採ると，それぞれのマスター・データ・セットに異った名前を与えてカタログせざるを得ない．それぞれのデータ・セットに特有な名前を与えてカタログすることは，重複した名前を与えないように配慮する必要があることから，そう簡単なことではない．それに，毎月，12ケ月前のマスター・データ・セットをカタログから削除する問題もある．このようなわずらわしさを排除するために，カタログ・システムでは相互に関連するデータ・セット群に同一の名前を与えて総称するような方法を採用したのである．これが，**世代別データ群名**である． COMML. PAYROLL. MASTER がこれに当る．ところで，ある世代と別の世代を区別するのに，4桁の**世代番号**が割り当てられる．**第13図**に示した G0001, G0002, G0003, G0004 などがこの世代番号である．さらに，世代番号に付加えてバージョン番号が与えられる．このバージョン番号は，ある種の世代のデータ・セットが特別な事由で更新されたような場合に役立つ．第13図の1ケ月目に給与変動があって，第2世代のデータ・セットが更新されたようなケースである．この場合，世代番号はそのままで**バージョン番号**がV00から V01 に変る．それゆえ，バージョン番号は，世代番号の添字的役割りを果す．なお，世代番号，バージョン番号ともにデータ・セット・ラベルの1部に記録され，データ・セットがオープンされたとき，データ・セット名の一部を構成するものとみなされて比較の対象となることは，「ボリューム・ラベルとデータ・セット・ラベル」の項で説明した通りである．

世代番号は，第13図の例からも理解できるように，初代の世代のデータ・セットからどれだけ経過した世代であるかを示し，バージョン番号は，その世代のデータ・セットが何回目の更新を受けたものであるかを示す．特定世代の新

版（バージョン）は，新版の番号をもった世代番号を指定することで自動的にカタログに組入れられる．この処理で旧版の記入項目は自動的にカタログから削除されてしまう．

なお，GxxxxVyy の形式をデータ・セット名に付加したものを**絶対世代名**と呼び，その例は，**第14図**の左側に示した通りである．したがって，DD ステートメントの DSNAME のパラメーターでこのような形式で特定データ・セットを指定することにより，世代別データ群の中から該当するデータ・セットをつきとめることができる．

しかし，世代が多くなってくると G0035V03 なんていう形式で DD ステートメントを作成することは，わずらわしくもなり，かつ，各世代ごとのデータ・セットの対比表を作成しておく必要があるというやっかいな問題にもなってくる．そこで，もっと簡単な方法を使って，DD ステートメントで特定世代を指示するために**相対的世代名**を使うことが出来るようにしている．相対的世代名では，何時でも現世代，つまり，最新の世代を基準にする．最新の世代は，COMML. PAYROLL. MASTER（0）という表現をとる．そして1世代前は（−1），その前は（−2）という表記をする．逆に，次に作成すべき世代は，（＋1）という表現をとる．その例を，**第14図**の右側で示している．

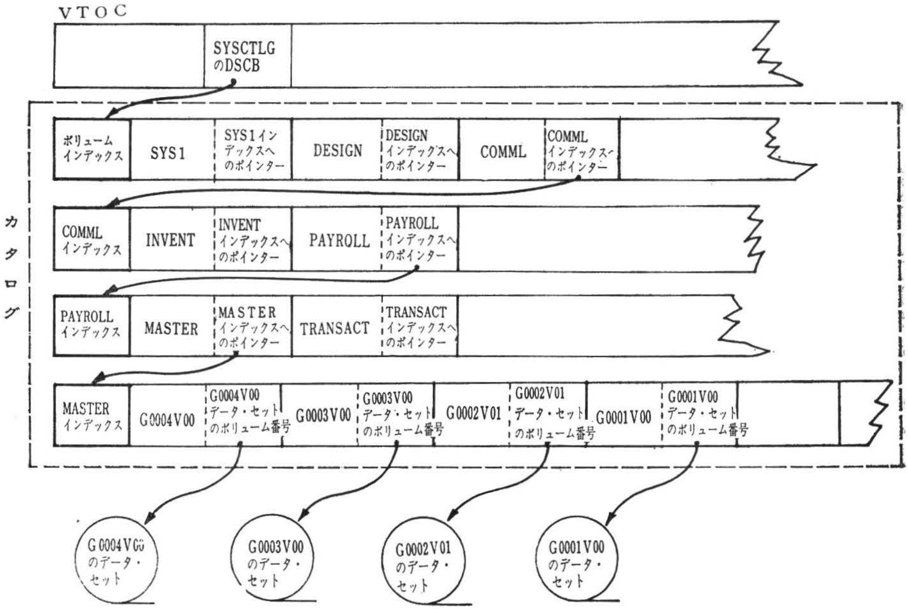
ある世代をカタログするのに相対的世代番号を用いると，カタログ管理ルーチンが自動的に絶対世代番号に変えてくれる．ただし，その場合のバージョン

	絶対世代名	相対的世代名
①	COMML. PAYROLL. MASTER. G0001V00	COMML. PAYROLL. MASTER(−3)
②	COMML. PAYROLL. MASTER. G0002V00	・・・・・・・・・・・・・・・・・
③	COMML. PAYROLL. MASTER. G0002V01	COMML. PAYROLL. MASTER(−2)
④	COMML. PAYROLL. MASTER. G0003V00	COMML. PAYROLL. MASTER(−1)
⑤	COMML. PAYROLL. MASTER. G0004V00	COMML. PAYROLL. MASTER(0)
⑥	COMML. PAYROLL. MASTER. G0005V00	COMML. PAYROLL. MASTER(＋1)

第14図　世代別データ・セットの参照名の例

番号は必ずV00となる．なお，割当てられる世代番号は，目下の最新の世代番号と指定された相対番号の双方を勘案して，適切な世代番号を決めてくれる．ともあれ，給与マスターの場合，毎月新しい世代が作り出されるので，他の世代は，1ケ月づつ古くなる．つまり，更新前に−1，−2，−3とみなされていた世代は，更新後には−2，−3，−4といった相対番号に変る．このような更新を管理してくれるのも，カタログ管理ルーチンである．

ところで，世代別データ群をカタログするインデックスであるが，**第15図**に示したように最下位のインデックスだけは，これまでみてきたインデックスと異る．これが**世代用インデックス**である．MASTER インデックスにたどりつくまでは，単一データ・セットの場合と全く同じである．単一データ・セットの場合だと，MASTER の記入項目の一部としてボリューム番号が記入されるのであるが，世代用インデックスの場合には，世代毎の世代番号とバージョン番号に対応してボリューム番号が併記される．ただし，カタログには1世代1データ・セットしか登録されないので，新しいバージョンのデータ・セットは

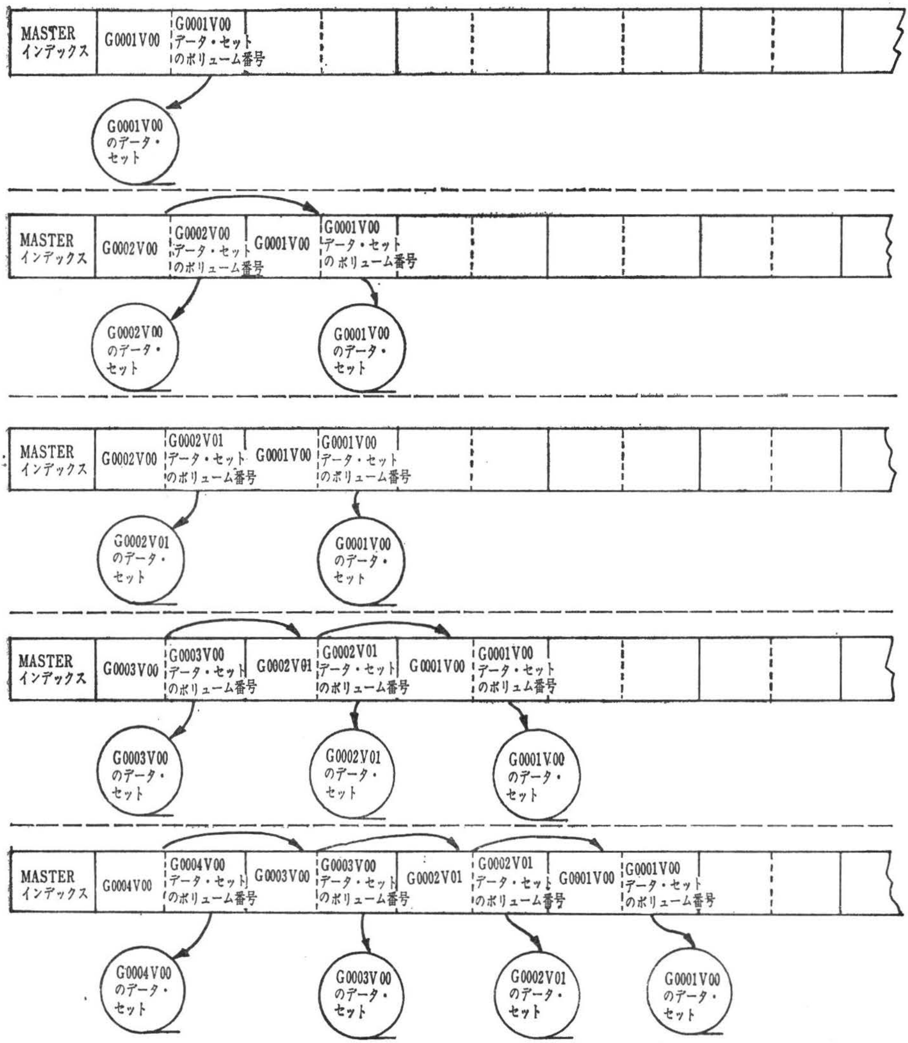


第15図　世代別データ群のカタログ

残らず最新バージョンのものだけがカタログされる．世代別データ群のカタログは，このように特別な世代用インデックスをもつので，データ・セットをカタログする前に，SYSCTLG データ・セットに世代別データ群用のインデックスを作り上げておく必要がある．これに使うのは以前にふれたIEHPROGM ユーティリィティー・プログラムである．これを使えば，世代別データ群の中におさめたい世代数だけの記入項目を持たせた世代用インデックスを作ることができる．給与マスターの1ケ年のデータ・セットを世代別データ群とする場合には12個の記入項目を作ればよい．また，このプログラムを使って，世代用インデックスがいっぱいになったときの旧世代のデータ・セットの処理の仕方を指示することもできる．その1つは，インデックスがいっぱいになると，旧世代のすべてのデータ・セットをカタログから取除いて，新世代のシリーズをカタログするやり方である．たとえば，1年分のデータ・セットがカタログされると，翌年1月に新しい世代をカタログするときに過去一年分のデータ・セットを廃棄してしまうやり方である．もう1つの方法は，一番古い世代だけをカタログから取除いて，その代りに新しいデータ・セットをカタログに入れるやり方である．この方法だと，いつでも過去1ケ年分の給与マスターがカタログに残ってゆくことになる．いずれの方法を採っても，その管理は，カタログ管理ルーチンが自動的にやってくれる．

ユーティリィティー・プログラムで世代用インデックスまで作りあげておくと，あとは個々のデータ・セットを作成するごとに，DD ステートメントを通してカタログすることができる．DISP=(NEW, CATLG) としておけばよい．新世代のデータ・セットがカタログされるごとに，世代用インデックスの記入項目は1つづつ右に移されることによってカタログされる．**第16図**にその様子を示しておいた．注意すべきは，この図の3段目で示したように同一世代で新しいバージョンのデータ・セットがカタログされる場合には，記入項目の移動を伴わない．したがって古いバージョンのデータ・セットは自動的にカタログから削除されてしまうことになる．

ある世代をカタログを通してつきとめる方法は，単一データ・セットの取出しの場合と同じである．その場合には第14図に示したような絶対世代名か相対



第16図　新世代データ・セットの作成による世代用インデックス記入項目の移動

的世代名のいずれかを使用する必要がある．もしも，相対番号や絶対番号をつけないで， COMML. PAYROLL. MASTER とするとカタログされている全世代のデータ・セットが入力データ・セットとなってしまう．ただし，このような扱いができるのは，世代別データ群を構成するすべてのデータ・セットが同一のデータ・セット編成法と属性をもつているときに限られる．

以上，世代別データ群の構成とカタログについて説明してきたが，このように関連するデータ・セットのグループ化を計ることの利点は， 1）グループ内のすべてのデータ・セットを共通の名前で取扱える， 2）制御プログラムが自動的にグループ内のデータ・セットの相互関係を管理してくれる， 3）また，好みの方法にしたがって古い，不要になった世代を自動的に削除してくれる，ことなどにある．

4.　システム・データ・セットの構成とカタログ

カタログの対象になるデータ・セットは，単に普通のデータ・ファイルだけではない．プログラム・ライブラリーと呼ばれる区分データ・セットであってもカタログすることができる．したがって，オペレーティング・システムを構成するうえで欠くことのできないデータ・セット，つまり，**システム・データ・セット**もカタログされたデータ・セットである．

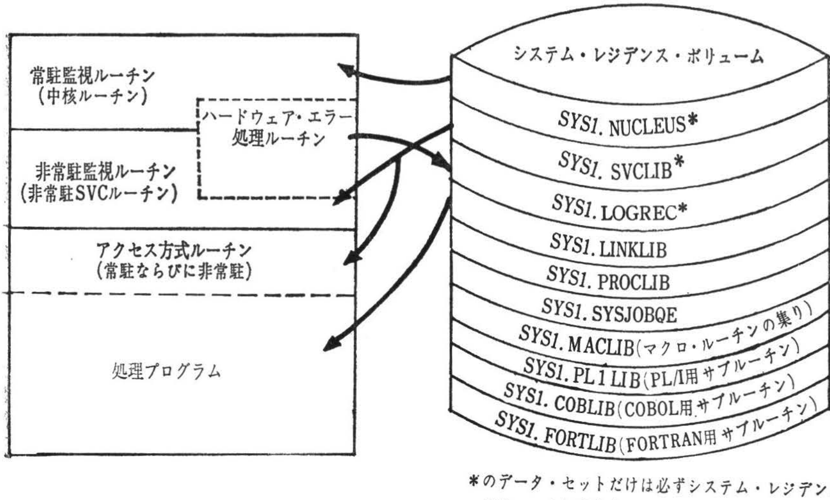
システム・データ・セットは，上位のインデックスとして必ず SYS1 を持つ．唯一の例外は， SYSCTLG である． SYSCTLG もシステム・データ・セットであるが，これだけは単一名である．SYSCTLG データ・セット自体がカタログに登録されることはないからである． SYSCTLG 以外の主要なシステム・データ・セットを図示したのが**第17図**である．

このうち SYS1. NUCLEUS, SYS1. SVCLIB, SYS1. LOGREC データ・セットは必ずシステム・レジデンス・ボリュームに収容する必要がある．しかし，他のシステム・データ・セットは，別のボリュームに収容することもできる．どのデータ・セットをシステム・レジデンス・ボリューム以外に配置するかは，システムの構成を考慮に入れたうえで一番効率のよい方法を工夫する必要がある．効率を考えた場合には，磁気ディスク装置だとアクセス機構の動きを

最少限にするようなシステム・データ・セットの配置ということになる．その配置の仕方は，全くユーザーに任されていることであり，**システム生成**（system generation）を通してその配置を決定することができる．

さて，それぞれのシステム・データ・セットの役割りを簡単に説明しておこう．SYSCTLG データ・セットはすでにくわしく説明してきたので今更説明はいるまい． SYS1. NUCLEUS は，第17図で図示したように監視プログラムのうちで常駐ルーチンが収容されるシステム・データ・セットである．このルーチンは，一般には**中核ルーチン**（Nucleus）と呼ばれるもので常時主記憶装置に存在する．制御プログラムの他の部分が必要になると，この中核ルーチンによって主記憶装置に呼び込まれるのである．中核ルーチンは，監視プログラムのうちでも使用頻度の高い機能の部分である．これを一括収容しているのが SYS1. NUCLEUS である．

SYS1. SVCLIB は，その言葉が示すように SVC 命令（監視プログラム呼び出し命令）を通じて利用されるルーチンを主体としたライブラリーである．し



*のデータ・セットだけは必ずシステム・レジデンスボリュームに存在すること

第17図　システム・データ・セットの種類とその機能

かも，主記憶装置に必要のつどロードされる SVC ルーチンをまとめて収容している．そのほか，アクセス方式ルーチンもすべて SYS1. SVCLIB に貯えられている．

監視プログラムとアクセス方式ルーチン以外のプログラム，たとえば，ジョブ・スケジューラーや連係編集プログラム，言語翻訳プログラム，ユーティリティー・プログラムあるいは分類・突き合わせプログラムなどはすべて SYS1. LINKLIB に収容される．このシステム・データ・セットにはユーザーの作成したプログラムも登録することができる．もちろん，ユーザー・プログラムは必ずしもこのライブラリーに収容する必要はなく，独自のプログラム・ライブラリーを作ることも許される．

SYS1. LOGREC には，入出力装置，制御装置，チャネルあるいは CPU 等で発生したエラーの分析内容や統計的データが自動的に集約される場所である．これらデータは，後刻，診断用の特別なプログラムで分析されハードウェアの故障の原因をつきとめたり，ハードウェア保守の貴重な資料となる．

SYS1. PROCLIB は，定形的なジョブに関する処理手順（ジョブ制御カードの集まり）を収めておく場所である．このライブラリーに収めた処理手順をジョブ制御カード上の簡単な指定で呼び出して利用することができる．詳細は「ジョブ管理の諸機能とその構造」の章で説明する．

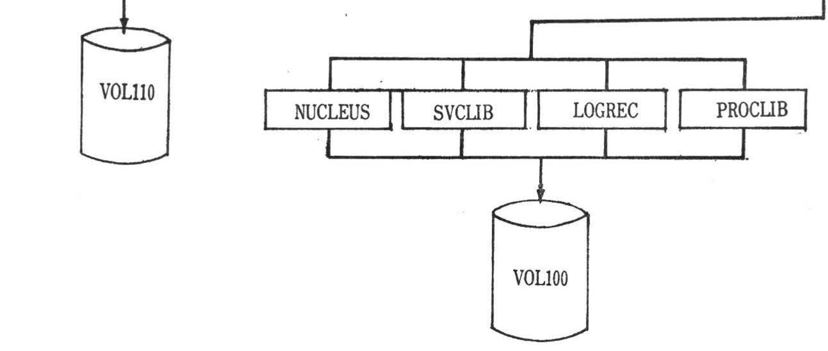
SYS1. SYSJOBQE は，ジョブ・スケジューラーが読込んだジョブ制御カードの内容を分析し，一定の形式に集約したブロックとして一時的に貯えておく場所である．JCB (Job Control Block)，JSCB (Job Step Control Block)，JFCB (Job File Control Block)，TIOT (Task Input/Output Table) などの形式として SYS1. SYSJOBQE に書き出され，ジョブあるいはジョブ・ステップの開始にあたって，これらの情報が活用される．

SYS1. PL1LIB，SYS1. COBLIB および SYS1. FORTLIB には，それぞれのプログラム言語で利用されるサブルーチンがまとめて収容されている．これらサブルーチンは，プログラムが連係編集される時点で自動的に組入れられる．

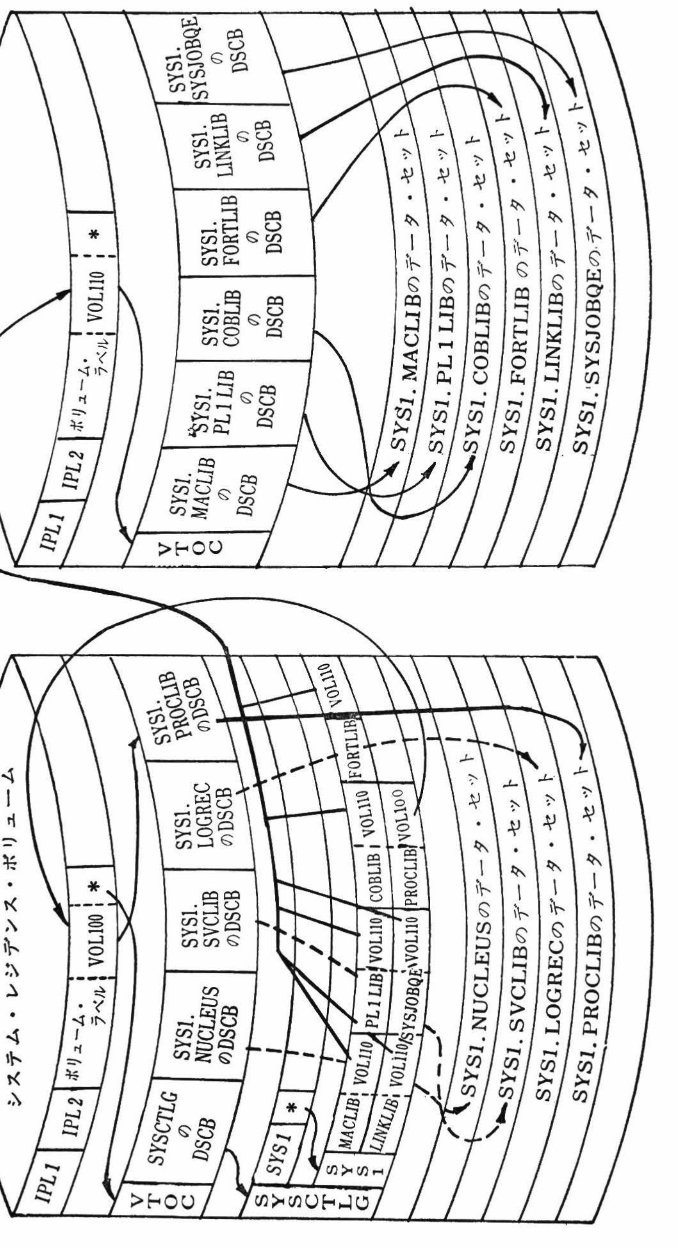
SYS1. MACLIB には，アセンブラー言語で使用するマクロ命令に対応する

ルーチンが収容されている．プログラムがアセンブルされるとき，マクロ命令で指定したパラメーターに見合った仕立て上げが行われたうえでプログラムに組込まれる．

これらシステム・データ・セットは，すべてカタログされたデータ・セットであることが要求される．すでに先に説明したように SYS1. NUCLEUS, SYS1. SVCLIB および SYS1. LOGREG を除いたシステム・データ・セットはシステム・レジデンス・ボリュームに同居する必要性はない．そこで，**第18図**に示したごとくシステム・データ・セットを２つのボリュームに分散収容することも可能である．システム・レジデンス・ボリュームには必ず収容すべきシステム・データ・セット以外に SYS1. PPOCLIB を収容するものとする．他のシステム・データ・セットはすべて別のボリュームに記憶させるものと仮定しよう．この場合，システム・レジデンス・ボリュームはボリューム番号 VOL 100 であり，他のシステム・データ・セットを収容するボリュームはボリュー



第18図　システム・データ・セットの樹木式構造の例



第19図　システム・データ・セットとそのカタログの探索

ム番号 VOL110 とする.

第18図のシステム・データ・セットのボリューム割り当てにもとづいて, カタログ構造を考えると**第19図**のようになる. SYSCTLG を除いたすべてのシステム・データ・セットがカタログに収容される. ボリューム・インデックスには SYS1 が登録され, すべてのシステム・データ・セットが SYS1 のインデックスの支配下におかれる. 本来システム・レジデンス・ボリュームに収容すべきデータ・セットは VTOC 内に収められた DSCB を通じて直接該当データ・セットの位置をつきとめることができる. しかし, SYS1. PROCLIB のように必ずしもシステム・レジデンス・ボリュームに同居させる必要のないデータ・セットは, カタログに収容された記入項目を通じてその場所が見出されることになる. ましてや, 他のボリュームに収容されているデータ・セットは, カタログされたインデックスを通してのみ, それぞれが収容されているボリュームを探し当てるより手がかりはないのである. 該当ボリュームが見つかればあとはボリューム・ラベルと VTOC を通して, 所要のシステム・データ・セットの記憶場所はつかめる.

このようにシステム・データ・セットといえども通常のデータ・セットと全く同じようにカタログ・システムのもとでの制御を受けるのである. カタログ・システムの管理をうけるのは単にデータ・ファイルだけでなく, プログラム・ライブラリー的区分データ・セットも同様である. したがって, 原始プログラム, 目的プログラムあるいはロード・モジュールを区分データ・セットとしたユーザー・ライブラリーを作り上げ, カタログ・システムのもとで管理してゆくこともできるのである.

六. パスワードによるデータ・セットの保護

近時, 世ではプライバシー侵害の問題がクローズ・アップされやかましく議論されてきた. 一方, 産業界では, 産業スパイの問題が取り上げられ, 企業の機密保持の対策が真剣に検討されるようになっている. ところで, この種の問題は情報処理システムと無関係な問題ではなくなってきている. プライバシー

の問題でいうなら，国民総背番号の問題がプライバシーを侵害する可能性があるとして，一時議論の的になったことがある．また，企業の機密盗用に関しては，ある計算センターを舞台にした購読者名簿のコピーの問題が現実に発生した．

いずれにしろ，コンピュータ・システムの利用が拡大し，国民にかかわりをもつ諸問題をもコンピュータの力を借りて記憶処理をする傾向が強くなればなるだけ，プライバシー保護という問題を一層真剣に考えざるを得なくなる．企業の機密保護の観点からすると，旧来のデータ処理が事務の事後処理に集中していたのに対して，最近では，経営戦略の確立や経営予測あるいは新製品の設計開発の仕事にまで拡大しているのであり，ほとんどの企業の機密的情報がコンピュータに集中しつつある．他方，従来のコンピュータ利用が集中的閉鎖的利用であったのに対し，現在は通信回線を利用したオンライン・システム，タイムシェアリング・システムへと開放的利用へと進んでいる．従来のようにごく一握りの専門要員だけのコンピュータの時代から，誰れでも自由に対話し，利用できるコンピュータへと発展してきている．このことは，誰れでもがその気になればコンピュータに集中された機密情報を取出したり，書き変えたりすることができることを意味している．それだけに一層プライバシー保護，機密情報の保護の問題が情報処理産業界あるいは情報処理システムの中心課題となってくる．

ところが，これまではせいぜいデータ・ファイルにつけられたラベルをもとに正しいデータ・ファイルを使用しているか，あるいは有効期限内にあるデータ・ファイルを壊す恐れはないかの検査をしている程度で，あとは人間による保護管理あるいは相互信頼の原則で処理されてきたのが実情である．コンピュータ・ハードウェアないしは，コンピュータ・ソフトウェアでの自動保護の工夫はさほどなされていなかったといえる．それに対して，オペレーティング・システムでは，完全な機密情報の保護といいきれるものではないが，許された人だけしか特定データ・セットを処理することができない機能を組込んでいる．これが**パスワードによるデータ・セットの保護**である．あるデータ・セットの取扱いを任された人にだけ，鍵に相当する合言葉を与えておく．そのデー

タ・セットが OPEN されようとするとその合言葉を打込まないことにはプログラム処理が続行しないような仕組みになっている．したがって，機密情報なりプライバシーに関係する情報なりの取扱いを委任されている人々以外が，特定データ・セットを処理することはできないのである．もちろん，時が立つにつれてパスワードが権利なき人々にも知れわたってしまう危険性が無いとはいえない．そうした危険性がある場合には定期的にパスワードを書き変えることで，ごく少数の許された人々だけのものとすることができる．

さて，このパスワードによるデータ・セットの仕組みがどうなっているか説明しよう．

該当データ・セットを保護するかどうかは，データ・セット・ラベルの中の情報で示す．磁気テープ・ボリュームのデータ・セットの場合には，HDR1 に**データ保護標識**という１バイトのフィールドがあったことを思い出して欲しい． DASD ボリューム上のデータ・セットでは，このフィールドが形式１の DSCB 内にある．いずれにしろ，データ・セットを作成するさいに，DD ステートメントの LABEL パラメータで PASSWORD と指定することによって，このフィールドに保護標識が立てられる．一方，保護されたデータ・セットの名前とパスワードを対比させた情報を， PASSWORD と名付けられた特別なデータ・セットに収容しておく．この PASSWORD データ・セットは必ずシステム・レジデンス・ボリューム上に貯えておく．データ・セットの編成は，順次編成であり，保護されたデータ・セットそれぞれに１レコードの情報がある．そのレコードの形式は，**第20図**に示したように，52バイトのキー部と80バイトのデータ部から成り立っている．キー部ではデータ・セット名とその

キー部		データ部		
データ・セット名（44バイト）	パスワード（8バイト）	利用回数（2バイト）	保護モード（1バイト）	ユーザー・エリア（77バイト）

保護モード：
X"00"：読取り保護だけ
X"01"：読取り／書込み両方の保護

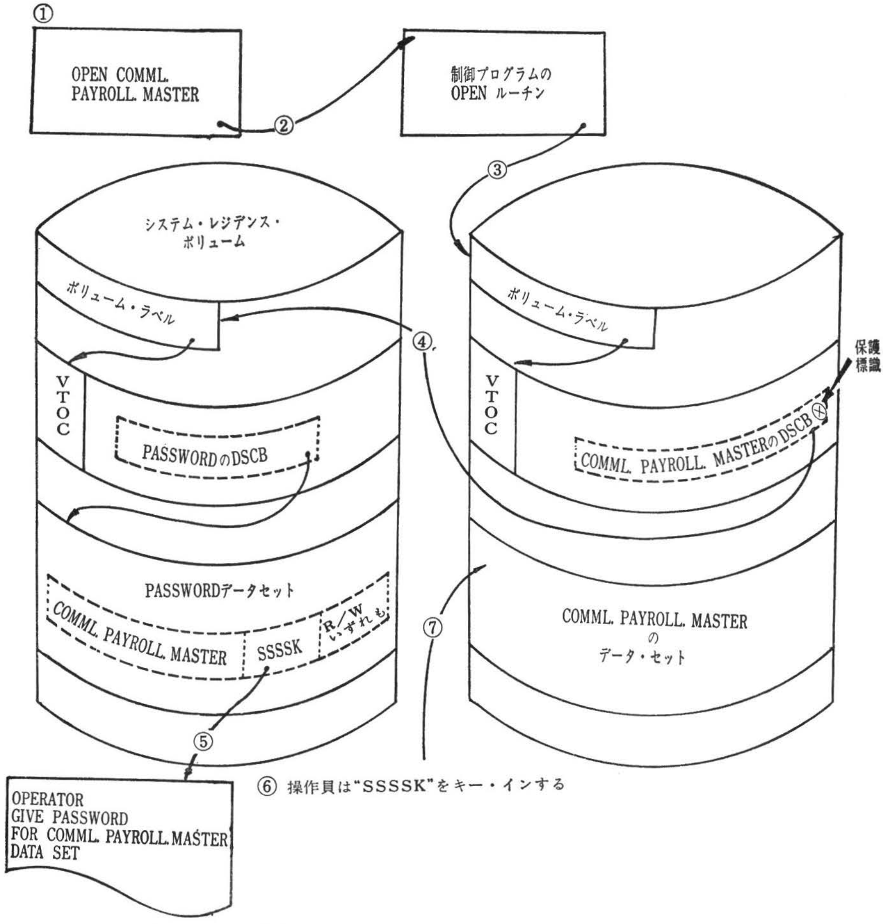
第20図　PASSWORD データ・セットの記入項目（レコード形式）

データ・セット用のパスワードが記入される．データ部の最初のフィールドには，該当データ・セットが利用されるごとに1が加えられてゆく．つまり，利用回数を示す．これを調べれば，許されざる人がこっそりこのデータ・セットを利用した可能性をつきとめることができる．1バイトの保護モードのフィールドでは，このデータ・セットの保護が，入力だけの場合か，更新を含めた出力の場合にも効果をおよぼすかを規定する．たとえば，給与マスター・データ・セットのような場合は読み書き両方を保護しておかないことには，心ない従業員が自分の基本給だけを都合良く書きかえてしまう恐れがある．77バイトのユーザ・エリアには，任意の情報をおさめることができる．

この PASSWORD データ・セットを作成し，保護データ・セットのために第20図のようなレコードを書き込む責任はユーザにある．ユーザーがこのデータ・セットを作成し保守するルーチンを書いておかなければならない．そして，SYS1. LINKLIB なりユーザー用のライブラリーにあらかじめ登録しておく．なお，この PASSWORD データ・セット自体が誰かにアクセスされて，その内容が印刷されてしまうと元も子もなくなるので，PASSWORD データ・セットそのものを保護データ・セットとするのが普通である．こうしておけば，オペレーティング・システムの制御プログラムがこのデータ・セットをアクセスするとき以外は，必ず合言葉をキー・インする必要が生じる．

データ・セット・ラベルにデータ・セットの保護標識を入れ，かつ，保護用データ・セットのためのレコードを PASSWORD に記録するとこれでデータ・セットの保護体制は出来上ったことになる．では，保護されたデータ・セットが利用される場合のステップはどうなるであろう．**第21図**でそれを示している．

処理プログラムの中で，データ・セットの OPEN が指示されると制御プログラムの OPEN ルーチンにコントロールが移る（第21図の①と②）．OPEN ルーチンでは，ボリューム・ラベルを通じて VTOC 内の該当データ・セットの DSCB をつきとめる（第21図の③）．データ・セットが，磁気テープ・ボリュームにあると HDR1 である．データ・セット・ラベルのデータ・セット保護標識のフィールドを調べ，該当データ・セットが保護データ・セットであるかど



第21図 保護されたデータ・セットの OPEN されるステップ

うか確認する．保護データ・セットでないと，OPEN ルーチンの次のステップに入る．保護データ・セットの場合だと，PASSWORD データ・セット内の対応レコードをつきとめる（第21図の④）．そのうえで，操作員に対してこのデータ・セットのパスワードをキー・インする旨のメッセージを出す（第21図の⑤）．操作員が，そのデータ・セットの取扱いを任せられている人なら，正しいパスワードをキー・インしよう．その任にない人だと，パスワードを知らない筈である．でたらめのパスワードは，PASSWORD データ・セット内のレコードに入っているパスワードと比較することによって，すぐに誤ったパスワ

ードであることが確認される．ただし，実際には2回までのキー・インが許されている．ともあれ，正しいパスワードが入力されると，OPENルーチンは，利用回数のフィールドを1だけ増やしたうえで，そのデータ・セットを利用可能な状態にする（第21図の⑥と⑦）．なお，正しいパスワードがキー・インされない場合にはそのプログラムの実行は中止されてしまう．したがって，一切のデータ・セットのアクセスはできないことになる．

以上のような方法によって，情報処理システムに集中する機密情報あるいはプライバシーに関する情報を，心ない人々から保護しようとしているのである．

七．直接アクセス・ボリュームのスペース割当て機能

「直接アクセス・ボリュームのスペース割当て機能」の説明に先立って，DASD のハードウェア的特性についてふれておこう．DASD のハードウェア特性がスペース割当ての機能を理解するうえで必須のものとはいえないが，DASD に関するデータ管理機能を十分に理解するうえでは欠くことのできないことである．

1. DASD のハードウェア特性

データ・セットの編成いかんにかかわらず，オペレーティング・システムのもとで作成されるデータ・セットのいずれも DASD ボリュームに記憶することができる． DASD 上に収容されたブロックあるいは物理レコードには，アドレスがつけられるのでどのレコードでも即座につきとめることができる．つまり直接アクセスが可能なのである．

DASD には，磁気ディスク記憶装置，磁気ドラム記憶装置あるいはデータ・セル装置などがある．これら装置は，その外観，記憶容量，データ転送速度などに相異があるものの，データの記録方式，データ検査の方式あるいはプログラミング上の点では機能的，論理的に変るところはない．各ボュームの記録面は多数のトラックに分かれており，同心円状として配列されている．トラック

の数や記憶容量は装置によって異るが，その機能や物理的構成は，いずれも後述するような内容となっている．

相互に関連するトラックの論理的配置は，記録面を基準にしているのではなく，シリンダー状を基準にしている．つまり，水平面でなく垂直面を基準にアドレス付けが行われている．これが，**シリンダー概念**と呼ばれるものであり，順次処理においてアクセス機構の無駄な動きを最少限にくいとめるようにしている．これによって，DASD が直接アクセスだけでなく順次処理にも向くようになるのである．

1） データの検査とデータ転送

ご存知のように CPU では，**パリティ検査**（parity check）を行っている．データの正確性を期するため，CPU 内でのバイトにはパリティ・ビットがつけられている．つまり，バイトが構成されるとき，バイト内のビットが奇数個になるようにパリティ・ビットが0ないし1にセットされる．そして，CPU でデータをアクセスするときはいつでもそのパリティが検査される方式である．

ところで，DASD 上のデータは，いつでも DASD 制御装置を経由して転送が行われる．そこで，DASD 上の記憶能力をより効率的に使うために，CPU のようにバイト毎にチェック・ビットをつけるのではなくデータ域ごとにチェック・ビットを対応させる方法をとっている．データが CPU から DASD 制御装置に転送されると各バイトのパリティ・ビットを取除いてしまう．その代り，**サイクリック・チェック**（cyclic）と呼ばれる2バイトの検査バイトを算出し，各データ域の最後に付加する．

DASD 上からデータが転送されると，制御装置では読みとったデータ域を調べ，そのデータ域に対するサイクリック・チェック・バイトを再計算したあとで，DASD 上に記憶されていたチェック・バイトと比較することによって検査をする．その結果，両者が等しくないと，**データ検査誤り**（data check error）の標識をセットする．これが**サイクリック・チェック方式**である．

つぎにデータの転送方法に簡単にふれよう．

チャネルと制御装置間では，データはバイト単位で転送される．入力の場合

だと，制御装置で奇数パリティ・ビットを各バイトに付加したうえでチャネルに送り出す．したがって，チャネルと制御装置間では，９ビット１バイトで同時に転送されるのである．

一方，DASDと制御装置内では，１ビットづつ転送される．制御装置では，入力の場合だとビット単位の逐次転送からバイト単位の平行転送に，逆に出力の場合だと平行転送から逐次転送に変換作業を行っている．

2)　DASD トラック上の形式

さて，DASD 上のデータ記憶の基本をなすトラックであるが，いずれの DASD でもインデックス・マーカー，ギャップ，ホーム・アドレス，レコード０，それに引きつづいてデータ・レコードがそのトラックの終りまで何個か記録されるという形式をとる（**第22図**）．以下でその詳細について説明しよう．

①インデックス・マーカー

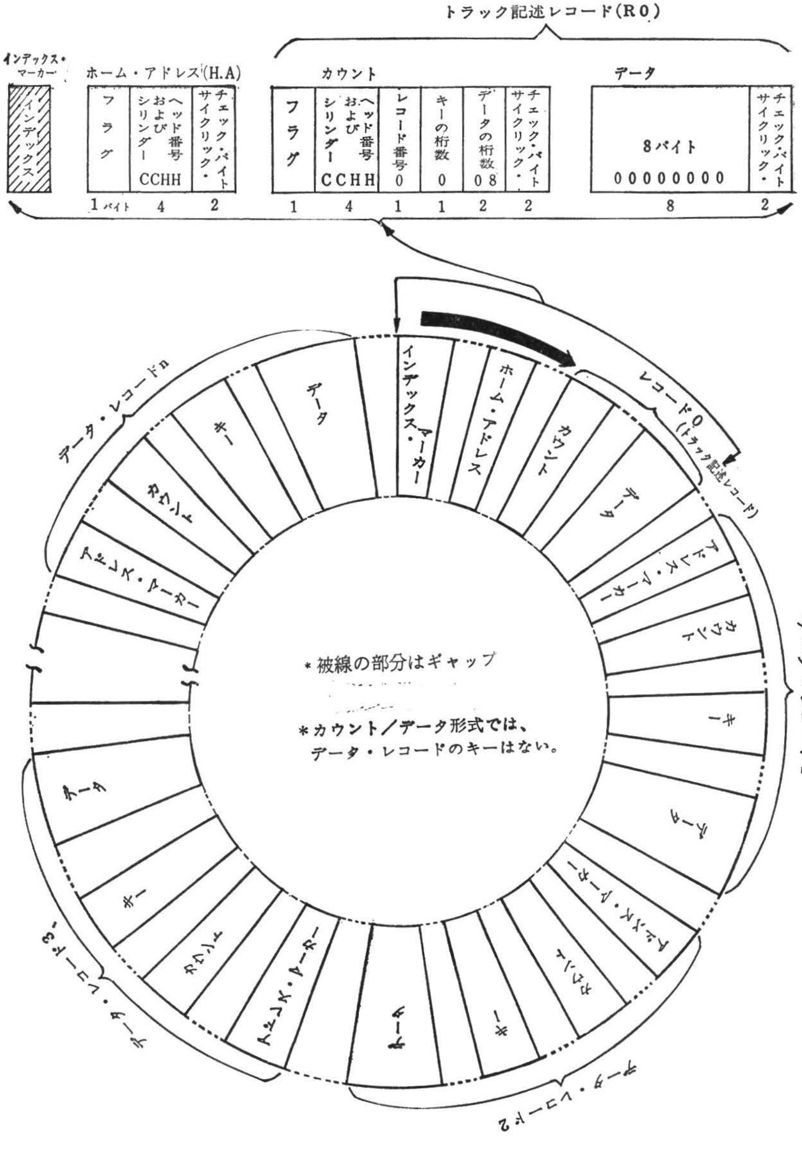
インデックス・マーカーは，各トラックの物理的な開始点を示す．実際には，１つの記録媒体（たとえば，ディスク・パック，ドラム）に１個のインデックス・マーカーしか存在しない．そして，その媒体上のすべてのトラックがこのインデックス・マーカーで同期化されるようになっている．

②ギャップ

ギャップは，トラック上の記憶域を他のそれと分離する．ギャップの長さ自体は，装置の種類やギャップが存在する位置，ないしはレコードの長さなどによって異る．

③ホーム・アドレス（第22図上側参照）

各トラックに１個のホーム・アドレスがある．インデックス・マーカーのつぎに位置し，７バイトのレコードである．最初５バイトで，そのトラックの状態と記憶装置内でのそのトラックの物理的位置を記述する．最後の２バ



第22図　トラック上のレイアウト

イトは，サイクリック・チェック・バイトである．

ホーム・アドレスの目的は，各トラックを識別する情報を与えるものであるが，何らプログラミング上には影響を与えない．装置内の各トラックはこのシリンダー番号とヘッド番号とでつきとめられる．ちょうど，大きな市に存在する特定の家をつきとめるのに町名（シリンダー番号に相当）と番地（ヘッド番号に相当）を使うのと同じである．普通のプログラム・オペレーションでは，このホーム・アドレスを利用することはあるが，その内容に影響を与えるようなことはない．このホーム・アドレスを CPU から転送できるのは“Write Home Address” という指令だけである．逆に，CPU にホーム・アドレスを転送するのは，“Read Home Address” という指令である．これら指令が使われるのは，後述する DASD 初期化用ユーティリィティー・プログラムである．

トラックの状態を示すのが，ホーム・アドレスの一部を構成するフラグ・バイトである．この**フラグ・バイト**は，“Read Home Address” 指令で自動的に読取られ，検査される．フラグ・バイトのビット0から5までは，いつも0である．ビット6では，トラックの状態を示す．“0”だと正常に作動するトラックを，“1”だと欠陥のあるトラックを示す．さらにビット7は，トラックの扱いを示す．“0”だと，本来のトラックを，“1”だと代替用トラックを示す．**本来のトラック**（primary track）とは，もともとデータを記憶すべきトラックであり，**代替用トラック**（alternate track）とは，本来のトラックに欠陥があることにより代りに利用されるトラックである．フラグ・バイトは，ホーム・アドレス以外にもレコード0，データ・レコードのカウント部分にもあるが，いずれもビット6と7は同じ扱いである．その具体的例は**第23図**に示している．

ここで，シリンダー番号とトラック番号について少しふれておこう．シリンダー番号は，データが記憶されている特定シリンダーの位置を示す2バイト（CC）であり，ヘッド番号は，そのシリンダー内の特定トラックを示す2バイト（HH）である．したがって，このシリンダー番号とヘッド番号を組合わせることによって，特定のトラックがつきとめられるのである．

ホーム・アドレスの最後にあるサイクリック・チェック・バイトの機能は先述した通りである．なお，ホーム・アドレスと次のレコード域とを分離するために自動的にギャップが作られる．

④レコード０（第22図の上側および第23図参照）

各トラックのホーム・アドレスに後続する最初のレコードが，レコード０であり，一般には**トラック記述レコード**と呼ばれている．このレコードは，本来のトラック上のある個所に欠陥がある場合，そのトラック全体を代りのトラックに移動することができるようにしたものである（第23図参照）．

以下にレコード０のカウント部の各フィールドとデータ部について説明する（第22図上側参照）．

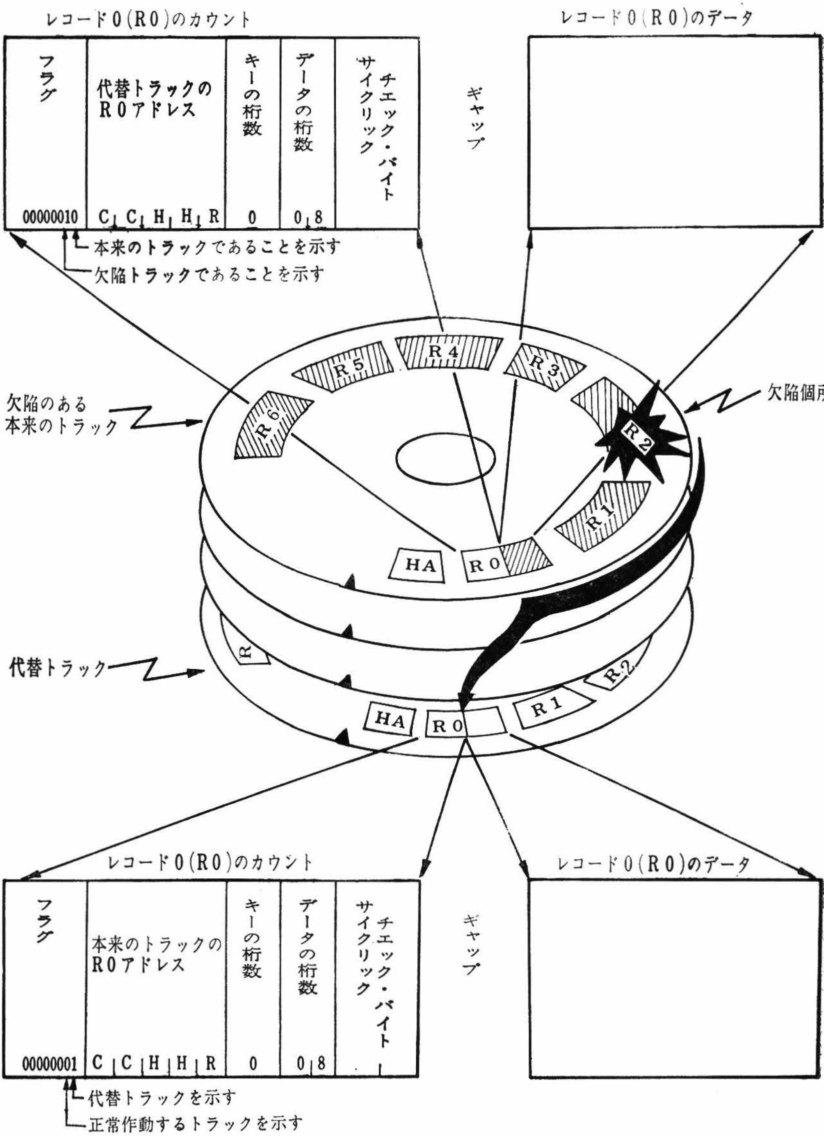
フラグ：このバイトは，カウント部の最初のフィールドである．このフラグの６と７のビットの機能は，ホーム・アドレスのフラグ・バイトで説明した通りである．

シリンダー番号：欠陥個所のない本来のトラックの場合には，このレコードが記憶されているトラックそのもののシリンダー番号が入る．しかし，このトラックに欠陥がある場合には，このトラックが移された代替トラックのシリンダー番号が入る．一方，代替トラックのこの場所には，レコードが移されたもとの欠陥トラックのあるシリンダー番号が収められる（第23図参照）．

ヘッド番号：欠陥のない本来のトラックだと，レコード０そのものが記録されているトラックに対応するヘッド番号が入る．もし，このトラックに欠陥個所があると，このトラックが移された代替トラックに対応するヘッド番号が入る．一方，代替トラックのこの場所には，レコードが移されたもとのトラックのヘッド番号が入る（第23図参照）．

レコード番号：この場所では，そのトラック上に存在するレコード０の位置を示す．レコード０の場合には，この番号は０である．

キーの桁数：そのレコードのキー部のバイト数が指定される．レコードがキーを持たないときは０である．レコード０はキーを持たないので，ここに



第23図　欠陥トラックが発見された場合の代替トラックの割り当て

は０が入る．

データの桁数：そのレコードのデータ域のバイト数が指定される．キーの桁数にしろ，データの桁数にしろサイクリック・チェック・バイトを除いたバイト数である．レコード０の場合には，データ域が８バイトであるので，このフィールドには０８が入る．

サイクリック・チエック：先述したようにデータ・エラーの発見に使われる．

データ域：この場所は，もともとオペレーティング・システムのもとで利用できるアクセス方式を規定するために設計されたものであるが，通常は０に初期化されてしまう．

⑤データ・レコード（第22図および**第24図**参照）．

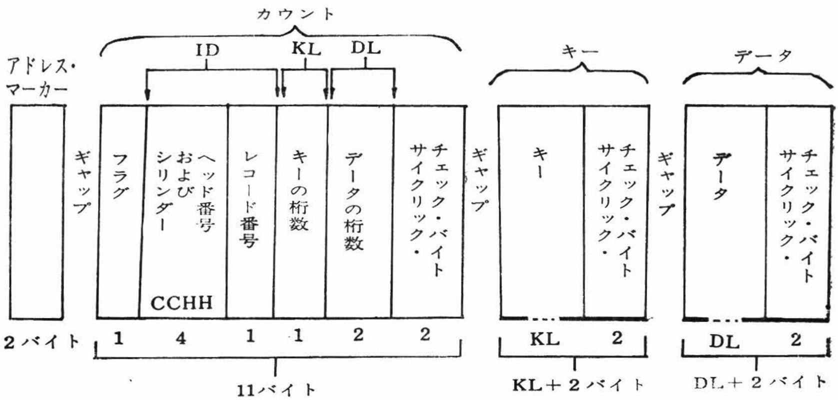
レコード０に引きつづいて，１つ以上のデータ・レコードがかかれる．データ・レコードには，キーの部分を持った形式とキー部分をもたない形式の２種類がある．キーとは，レコードを識別する主要な分類コードのことであり，たとえば在庫マスター・ファイルのような場合商品番号ないし部品番号などがこれに当る．要するに，特定レコードを検索する手がかりとなるものをキーとして使うことができる．特定シリンダーあるいは特定トラック上のレコードをアドレス（CCHHR）を使わないでキーで取り出すようなデータ・セットの場合には，カウント／キー／データの形式を採る．

以下にデータ・レコードの各フィールドについて説明する．

アドレス・マーカー：この２バイトで，各データ・レコードの始まりを示す．アドレス・マーカーは，レコードが書き出されるとき自動的に記憶される．このマークは，レコード検索，レコードの読み書きのオペレーションにおいて，レコードの始まりをつきとめるために使われる．

カウント部：この11バイトでは，後続するキーおよびデータ部について記述する．

最初のバイトであるフラグは，ビット０，１，６，７だけが意味があり，あとは０である．ビット６と７は，ホーム・アドレスの個所で説明したと同



第24図　データ・レコードの形式

じ機能をもつ．ビット0は，そのトラック上の偶数番目のデータ・レコードの場合には“0”で，奇数番目のときは“1”になる．このビットは，すべてのアドレス・マーカーが確実に存在するかどうかを DASD 制御装置でチェックするのに使われる．つまり，このビットが連続して同じ状態であると，アドレス・マーカーが存在しないものとみなされる．言いかえれば，データ・レコード全体が1個失われていることを示唆する．

ビット1は，後述するレコード・オーバーフローの特殊機能が組み込まれているときだけ使われる．オバーフロー・レコードでないときおよびオーバーフローしたレコードの最後のセグメントであるときは“0”，最後のセグメントを除いたオーバーフローのセグメントのときは“1”となる．

シリンダー番号，ヘッド番号は，そのデータ・レコードが存在するシリンダーとトラックを表示する．レコード番号は，該当データ・レコードがそのトラック上の何番目のレコードであるかを示す．

キーの桁数は，そのデータ・レコードにキー部が存在する場合のキーの長さを示す．一方，データの桁数では，データ部の長さを示す．いずれも，チェック・バイトの2バイトを除いた長さが示される．

キー部：キーには，先述したように部品番号だとか従業員番号のようにレ

コードを探し出す手がかりになるような情報を貯える．このキーを使って所要のレコードが見つけだされると，読み書きの命令が実行され，データ部の読み書きが行われるのである．キーを検索している間の比較は制御装置内で行われるので，検索中に CPU は他の命令を実行することができる．

データ部：この場所に，本来のデータが貯えられる．データ部には，ブロッキングしない形式で論理レコードを貯えることもできるし，ブロック化して貯えることもできる．ブロック化したときには，キー部にはブロックの最後の論理レコードのキーが挿入される．ブロックサイズは，システムの効率に大きく影響する．ブロックサイズが小さいと，無駄なスペース（つまり，ギャップ）が大きくなり，かつアクセス時間も増えてしまう．たとえば，80バイトのブロックだと，トラックいっぱいのブロックにくらべて35倍ものデータ転送速度を要するし，記憶装置の利用効率も半分以下に落ちてしまうほどである．したがって，レコードのブロック化が磁気テープの場合と同じように重要な要素となる．

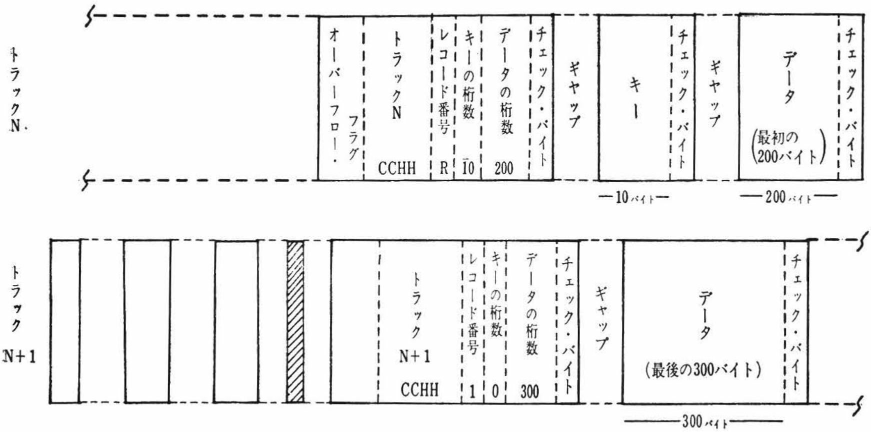
なお，データ部の最後にもやはり2バイトのサイクリック・チェック・バイトが付加される．

3) トラック・オーバフローの機能

トラック・オーバフローの機構が組込まれている装置で，かつ DD ステートメントか DCB マクロ命令（後述）でトラック・オーバフローの機能を指定することによって，ボリューム上の未使用のスペースを減少させることができる．この機能を使わない場合には，そのトラックの未使用のスペースに完全に入りきらないレコードはつぎのトラックの最初のレコードとして記録される．したがって，各トラックの最後には何らかの未使用のスペースができてしまう．ところが，トラック・オーバーフローの機能を使用すると，そのトラックに収容しきれないブロックが発生すると，そのトラックにはブロックの最初の部分だけを記録し，残りの部分はつぎのトラックにつづけてくれるのである．オーバーフロー・ブロックの各セグメントはカウント部を持ち，カウント部のデータの桁数のフィールドでは，そのセグメントの長さだけを示す．もしも，

そのブロックがキーを持つ場合には，そのキーは最初のセグメントにだけ書かれる．**第25図**に示したのがトラック・オーバーフロー・レコードの例である．

この例では，10桁のキーをもった500桁のレコードをトラックNに書き出そうとしたところ，そのトラックには200データ・バイトの空きしか残っていないので，残り300バイトを別のセグメントとしてつぎのトラックにオーバーフローさせている．これが，トラック・オーバーフローの機能である．



第25図　トラック・オーバーフロー・レコードの処理

なお，トラック・オーバーフローによってあるブロックがシリンダーからつぎのシリンダーの最初のトラックにまたがることはできるが，不連続のエクステントが割当てられた場合，あるエクステントからつぎのエクステントにトラック・オーバーフローすることはできない．

4)　トラック・アドレッシングの指定

DASD ボリューム上にデータを記録したり，逆にデータを取り出したりするのに使われるアドレスには，２つのタイプがある．２つのタイプがあるというよりも，オペレーティング・システムのデータ管理の機能のもとでは，２つのアドレスの仕方が許されているといった方が妥当であろう．そのアドレスと

は，実際アドレスと相対的アドレスである．**実際アドレス**とは，DASD ボリューム上のデータ・レコードのカウント部にあるアドレスと同じものでアクセスする方法であり，**相対的アドレス**とは，そのデータ・セットに割当てられた最初のトラックから何番目のトラックの何番目のレコードであるとか，最初のブロックを起点として何番目のブロックであるかの指定を行なう方法である．実際アドレスを使用する場合の利点は，相対的アドレスの場合には実際アドレスに変換するために時間を要するが，その時間を必要としないことにある．以下に実際アドレスと相対的アドレスについて簡単に説明しておこう．

①実際アドレス（actual address）

実際アドレスの形式は，MBBCCHHR である．Mは1バイトの2進数で，データ・エクステント・ブロック（Data Extent Block：略して DEB）の記入項目の相対的位置を示す．DEB は，そのデータ・セットがオープンされるときに自動的に作られる．各エクステントの記入項目は，そのデータ・セットに割当てられた連続するトラック群を記述する．つまり，Mはそのデータ・セットに割当てられたエクステントの順番を示すといってよい．

BBCCHH は，6バイトの2進数であり，該当レコードの存在するセル番号（BB），シリンダー番号（CC），ヘッド番号（HH）を指定する．これが**トラック・アドレス**である．なお，セル番号はデータ・セル装置についてだけのものであり，他の装置ではゼロである．

Rは，1バイトの2進数で，そのトラック上の相対的なレコードの順番を示す．これは，カウント部のレコード番号と同じである．

このような実際アドレスを使用した場合には，そのデータ・セットは，**移動不可**（unmovable）の取扱いをうける．移動不可とは，他の DASD 上の記憶域に移しかえることができないという属性である．なぜなら，実際アドレスを使っているので，そのままで再配置するとアドレス自体が狂ってしまうからである．それに，実際アドレスを使うと“装置からの独立性”が失われてしまうのも事実である．DASD 装置は，その記憶容量，シリンダーの数，トラック

の数などが装置の種類によって異るからである.

②相対的アドレス（relative address)

相対的アドレスには，相対的ブロック・アドレスと相対的トラック・アドレスの2種類がある.

相対的ブロック・アドレスは，そのデータ・セットを構成する最初のブロック（物理レコード）を基準に該当ブロックが何番目に位置するかを3バイトの2進数で指示する方法である.

一方，**相対的トラック・アドレス**は，TTR の形式である．TT は，2バイトの2進数であり，そのデータ・セットに割当てられた最初のトラックを基準に該当トラックが何番目のトラックに位置するかを示す．したがって，最初のトラックのそれは，00 である.

Rは，1バイトの2進数で，トラック上の最初のブロック（物理レコード）を基準に何番目のブロックであるかを示す．トラック上の最初のブロックは，0である．したがって，カウント部のレコード番号と較べて1だけ少ない数である.

相対的ブロック・アドレスにしろ，相対的トラック・アドレスにしろ，これを実際番地に変換するのはデータ管理プログラムの機能である．したがって，処理プログラムでは，相対的アドレスだけに関知することで事足りる.

2.　スペースの割当て要求とスペース管理ルーチン

直接アクセス・ボリュームの容量は，装置の種類によって異るが，大きなものだと何億バイトといった情報量にも達するようになっている．したがって，1つのボリュームに何種類ものデータ・セットの収容が可能となる．しかも，データ・セットを構成するレコードには追加・削除がつきものである．データ・セットそのものも有効期限を過ぎると削除される運命にある．削除の結果生じたスペースを他の新しいデータ・セットに割り振らないことにはスペースに無駄が生じてしまう.

こうなってくると，直接アクセス・ボリューム上のスペースを人手の介入で

管理することは極めてわずらわしい仕事であるだけでなく，スペースの有効利用という観点からみても得策ではない．ということから，本章の第4節「ボリューム・ラベルとデータ・セット・ラベル」の項で詳述したような DSCB を利用して，システム自体がスペースの割当てと自動管理を行ってくれる．

とはいうものの，スペースの割当てにさいして，特定データ・セットにどの程度の大きさのスペースを必要とするかをオペレーティング・システムに自動的に判断させることは無理である．何らかの方法で必要となるスペースを算出させるための基準を教えてやることが必要である．そのために，ジョブ制御ステートメントを通じて， DASD データ・セットが必要とするスペースの要求を出す．そこで，データ管理プログラムの一部をなす DASD **スペース管理ルーチン** (DASD space management routine) は，ジョブ制御カードを通じて要求されたスペースに関する情報と DASD ボリューム上の VTOC に貯えられているスペースに関する情報を突き合わせる手法で，スペースの自動割当ての機能を遂行してくれる．

本節では，まずどのような方法でスペースの要求が行われるかを検討したうえで，スペース管理ルーチンがそれに対してどのように機能するかをみてゆく．

1) DASD スペースの割当て要求

直接アクセス・ボリュームに新たなデータ・セットを書き出すときには，そのデータ・セットに必要と思われるスペースをジョブ制御カードの DD ステートメントのパラメータで指示する．単にスペースの大きさだけでなく，そのデータ・セットを収容する装置のタイプも指定できる．それにしたがって，DASDスペース管理ルーチンは，装置を選定し，所要のスペースを確保してくれる．この機能によって，より柔軟性のある DASD 上の記憶容量の利用が実現できるだけでなく，効率的なスペース管理という仕事の責任とわずらわしさから解放されることにもなる．

ところで，スペースの割当て要求の説明に入るまえに，**直接アクセス・ボリュームの初期化** (DASD initialization) について簡単に説明しておこう．直接

アクセス・ボリュームをデータ・セットの記憶に利用する前には，必ず初期化という仕事がつきまとう．つまり，直接アクセス・ボリュームを入手して最初に利用するに先立って，初期化を行う．この初期化を行うのに使われるのが，DASDI と呼ばれるユーティリティー・プログラムである．この DASDI プログラムを実行することによって，つぎの機能が果される．

- 80桁の標準ボリューム・ラベルを作りあげ，そのボリュームのシリンダー0，トラック0に書きだす．
- VTOC の初期化をする．VTOC を収容する場所と大きさは，制御カードで指定する．
- 各トラックにホーム・アドレスとレコード0を書きだす．
- 各トラックが正常に作動するかどうかを検査し，欠陥のあるトラックに対しては，代替用トラックの割当てを行う．

このような初期化が一たん行われてしまうと，そのボリューム上のスペースの割当て確保と管理の機能は，完全にスペース管理ルーチンに委ねることが可能となる．

さて，スペースの割当て要求であるが，その要求は，シリンダー単位，トラック単位あるいはブロック単位にでも行うことができる．ただし，装置の種類にこだわらないで（つまり，DASD の装置のタイプから独立して）スペースを要求したい場合には，ブロック数でスペースを指定する必要がある．なぜなら，シリンダー内のトラックの数だとか，1トラック内に収容できるブロックの数は，装置のタイプで大幅に異るからである．トラック単位やシリンダー単位でスペースを要求する場合には，トラックやシリンダーの記憶能力に十分なる配慮が必要であることはいうまでもない．

実際にスペースを要求する方法には，つぎの4種類がある．

- 特定のトラックを起点として，何トラックかのスペースを要求する．
- スペースとして必要な容量を，シリンダー単位，トラック単位あるいはブロック単位で要求し，その容量にふさわしい大きさをシステムに自動的に確保させる．
- 別のデータ・セットとシリンダーを分割して確保するようにシステムに要求

する.

●先行のデータ・セットに割当てられたスペースの一部を要求する.

以上4つの要求の仕方をまとめたのが**第26図**である. 第1の方法は, 容量と相対トラック・アドレスを指定するやり方である. 数量と書いたパラメータで必要なスペースをトラック数で指示し, アドレスの個所で相対トラック・アドレスを示す. **相対トラック・アドレス**とは, ボリューム上の最初のトラックから始まって何番目のトラックかを数字で示す方法である. いずれにしろ, この方法は, ボリューム上の特定位置に依存関係をもつデータ・セットに対してのみすすめることのできるテクニックである. なぜなら, この方法だと極めてスペースの割当てに関して柔軟性を欠いてしまうからである. もし, 要求するトラックがすでに別のデータ・セットで占有されていると, その仕事は実行できなくなってしまう. なお, ディレクトリーの部分では, 区分編成あるいは索引順次編成のデータ・セットにおいて登録簿ないしは索引の大きさを指定する.

SPACE=	(ABSTR, (数量, アドレス〔, ディレクトリー〕))
SPACE=	({TRK CYL ブロックサイズ}, (数量[, 増分 ,][, ディレクトリー , 索引]) [, RLSE ,]{, CONTIG , MXIG , ALX , }〔ROUND〕)
SPLIT=	{n (n, CYL, (数量〔, 増分〕)) % %, ブロックサイズ, (数量〔, 増分〕))}
SUBALLOC=	({TRK CYL ブロックサイズ}, (数量[, 増分 ,]〔, ディレクトリー〕, ステップ名.dd名)

注:

・{ } はこの中から1つだけ選択する

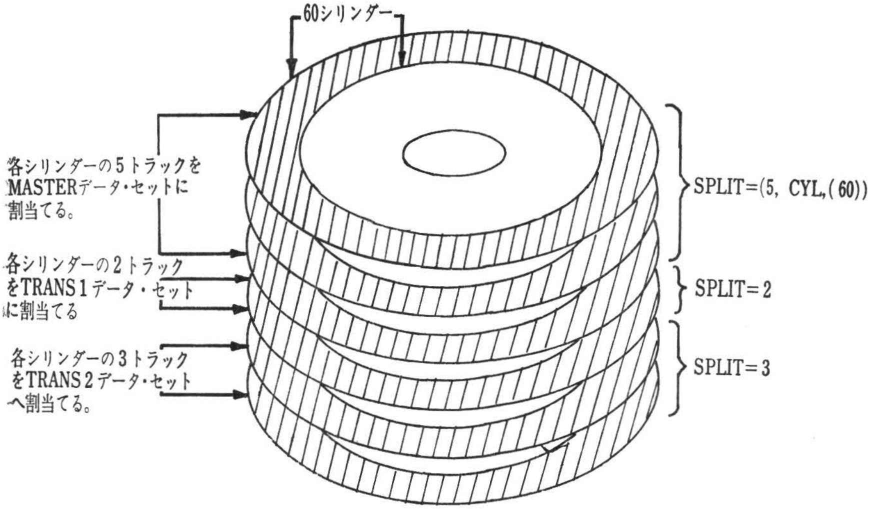
・〔 〕任意指定: 2行以上の場合は, 1つだけ選択するか全然選択しない.

第26図 DDステートメントにおけるスペース割当て要求のパラメータ

最も汎用性の高いスペース割当て要求の方法は，必要なスペースの容量を指定して，あとはスペース管理ルーチンにその割当てを任すやり方である．要求するスペース量は，トラック単位（TRK と指定），シリンダー単位（CYL と指定）あるいはブロック単位（ブロックの長さをバイト数で指定）のうちから，最適なものを選んで指示できる．必要なトラック数，シリンダー数あるいはブロック数は，数量の個所で指定する．要求したスペース量を追加レコードなどによってオバーする可能性のあるデータ・セットに対しては増分の個所で余分のスペースを要求しておくこともできる．こうすることによって，元のスペースがレコードで一ぱいになるたびに，同一ボリューム上に増分で指定したスペースだけを追加割当てしてくれる．また，区分編成や索引順次編成のデータ・セットでは，登録簿（ダイレクトリー）や索引の大きさも指定できる．ただし，索引順次編成のデータ・セットでは，増分の値を指定することはできない．そのほかのパラメータでは，デート・セットがCLOSEされたときそのデータ・セットに割当てたスペースのうち未使用の部分を解放したり（RLSE），スペースの連続性の制約条件をつけたり（CONTIG, MXIG，または ALX），確保したスペースがシリンダーの最初のトラックで始まり，シリンダーの最後のトラックで終るように要求する（ROUND）ことができる．

3番目のスペースの要求手法は，確保したシリンダーに関連するいくつかのデータ・セットを分割する手法である．2つ以上のデータ・セットが相互に対応するレコードを持っているような場合で，アクセス・アームの移動を最少限におさえることによって処理効率を上げたいような時には，シリンダー分割の手法を用いる．この方法では，割当てられる各シリンダーの何割かのトラックが各々のデータ・セットに与えられる．

シリンダーを2つ以上のデータ・セット間で分割するには，関連するDDステートメントをジョブ制御ステートメントの中に順番に並べる必要がある（**第27図参照**）．この順序内の最初のDDステートメントで，すべてのデータ・セットに必要なスペースの総容量（第27図では60シリンダー）を指定する．それ以降のDDステートメントでは，全体のスペースの中でのそれぞれの部分だけを指定するだけでよい．




```
// DD1   DD   DSNAME=MASTER,   SPLIT=(5,CYL, (60)),………
// DD2   DD   DSNAME=TRANS1,   SPLIT=2,………
// DD3   DD   DSNAME=TRANS2,   SPLIT=3,………
```

第27図　シリンダー分割の概念とその要求

直接アクセス・ボリューム上のスペースを要求する第4の方法は，**細分割当て**（suballocation）**の手法**である．この手法を用いると，直接アクセス記憶装置上の連続した場所に，いくつかのデータ・セットを順番に入れることができる．これを行うためには，まずあるデータ・セットを定義し，2番目に示したスペース要求の手法で連続配置したいデータ・セットのすべてが収容できるだけのスペースを確保する．細分割当てを目的としてスペースを確保するのであるから，スペース全体が連続した場所であるように（CONTIG パラメータを指定）しておかねばなねない．次に同一ジョブ内の後続の DD ステートメントで，元のデータ・セットに割当てたスペースの一部を SUBALLOC パラメータで要求する．SUBALLOC で要求したスペースは，元のデータ・セットから未使用のスペースが割当てられることになる．その指定方法は，第26図にまとめてある通りである．

以上4つの要求手法で，直接アクセス・ボリューム上のスペースの割当てを

指示することができるが，いずれにしろ，あらかじめデータ・セットが必要とするスペースの見積りを行っておくことが要請される．データ・セットに必要となるスペースを見積るには，装置のタイプ，トラックの記憶容量，シリンダー当りのトラック数，ボリューム当りのシリンダー数，データの長さ（つまり，ブロックサイズ）あるいはキーの長さなどを考慮する必要がある．

2）　スペース管理ルーチンの働き

DASD スペース管理ルーチンは，いうまでもなく直接アクセス・ボリューム上のスペースの割当て管理機能を果す制御ルーチンである．ここでは，このルーチンがどのような方法でスペースの管理を行うかを説明する．

スペース管理ルーチンは，VTOC を活用してこの機能を果す．VTOC は，前述したように DASDI ユーティリィティー・プログラムによって，全ての直接アクセス・ボリューム上の1部に作られる．この VTOC に，そのボリューム上の各データ・セットのDSCBおよびすべての未使用のスペースに対するDSCB が収容されている．スペース管理ルーチンは，新しいデータ・セットに対するDSCB を作り上げたり，削除されたデータ・セットの DSCB を取除いたりして VTOC の保守に当たる．データ・セットが作られたり，すでに存在するデータ・セットが拡大されたりすると， VTOC の中に収容されている適切な DSCB を探索して，未使用のスペースを見つけ出し，新しいデータ・セットや拡張されたデータ・セットに割当てる一方，そのスペースを空きのスペースから取除く．逆に，あるデータ・セットがボリューム上から削除されたとなると，そのデータ・セットの DSCB を VTOC から追放し，その DSCB 内で示されているスペースを空きのスペースとして返上する．また，スペース管理ルーチンは，あるデータ・セットに割当てられたスペースの未使用の部分をその DSCB を更新することによって，空きのスペースとして返上することもある．

いずれにしろ，VTOC 内のすべての DSCB に記述されている総トラック数は，何時でもそのボリューム上の総トラック数に一致するように保守しているのが，スペース管理ルーチンである．ところで， VTOC とそれを構成する DSCB については，本章の第4節「DASD のボリューム・ラベルとデータ・

セット・ラベル」の項で詳述しておいたが，ここで，DSCB の種類について再度まとめておこう．

- 形式0：DSCB として使用されていない VTOC 内の空きの 140 バイトのエリア．
- 形式1：データ・セットの主要な属性とそのデータ・セットに割当てられた3つまでのエクステントを記述．
- 形式2：索引順次編成のデータ・セットだけに使われる DSCB であり，形式1で示される属性以外のものを記述．
- 形式3：形式1では3つまでのエクステントしか記述できないが，これを越えるエクステントについて記述．
- 形式4：VTOC 自体のエクステントの記述．
- 形式5：該当ボリューム上の空きのスペースについて記述．
- 形式6：これについては「DASDボリューム・ラベルとデータ・セット・ラベル」の項で説明しきれなかった形式である．この DSCB は，シリンダー分割割当てにだけ利用される．この形式では，2つ以上のデータ・セットで分割利用されるスペースの範囲（つまり，エクステント）について記述するのである．先述したように，シリンダー分割割当てでは，必ず連続したスペースが割当てられる．したがって，これに割当てられたスペースは，その範囲全体が，形式5の DSCB から取除かれて，形式6の DSCB に1つのエクステントとして登録される．そのあとで，シリンダー分割のデータ・セットに割振られる．各々のデータ・セットに利用されたスペースは，各データ・セットの DSCB 内に記録される．そして，分割シリンダーのデータ・セットの全てが削除されるまで，全体のスペースが確保されたままになり，他に割振られることはない．分割シリンダーを構成する全レコードが削除されたとなると，この形式6の DSCB に記述されているエクステントははじめて取除かれ，形式5の DSCB に返却されるのである．

以上6つの形式の DSCB を巧みに駆使し，かつ，正しく更新することによって，直接アクセス・ボリューム上のスペースの割当てと管理を使うのがスペース管理ルーチンである．

データ・セットが作成される場合には，形式5の DSCB を検索して空きのスペースを探し出す．空きがみつかれば，必要なスペース量だけを，形式5の DSCB で記述されているスペースから差引く．一方，新しいデータ・セットに対しては，形式1の DSCB（必要なら形式2または3を含めて）を作りあげて，VTOC 内の所定の個所に記録する．また，データ・セットが削除されたとなると，そのデータ・セットの DSCB（つまり形式1，ないしは形式2と3）は，形式0の DSCB に置きかえられ，そのデータ・セットが占有していたスペースは，形式5の DSCB に完全に返上される．このような方法で，VTOC 内の DSCB を管理することによってスペース割当てを行っているのが，まさにスペース管理ルーチンなのだ．

スペース管理ルーチンの仕事は，単にそのボリューム上のデータ・セットに割当てられるスペースの管理だけではすまされない．VTOC 内の DSCB に利用するスペースに関しても管理責任が任されている．形式5や6の拡張に要するスペースにも考慮を払う必要がある．形式4には，VTOC 内のスペースの大きさが示されている．したがって，この管理ルーチンは，新たな DSCB を作るにさいしては，まずこの形式4をみて，新しい DSCB を収容するに足るスペースが VTOC にあるかどうか確かめる．それだけのスペースがなければ，古いデータ・セット，つまり有効期限を過ぎたものを削除したうえで，そのデータ・セットの DSCB を VTOC から取除くことによって VTOC 内のスペースを作り出すのである．このようにして，VTOC のスペース自体の管理責任を果す．

スペース管理ルーチンの見逃すことのできないもう1つの機能は，VTOC の完全性，保全性を遂行する機能である．ご存知のように，オペレーティング・システムのもとでは，多重タスクが可能である．同時に複数個のタスクが同一データ・セットをアクセス可能性がある．このような場合，一方のタスクで利用しているデータ・セットを別のタスクでも利用する可能性があるわけで，その結果，VTOC の内容が，どちらかのタスクによって変更されてしまう危険性がある．

たとえば，1つの CPU だけを利用するオペレーティング・システムのもと

でも，2つ以上のタスクで，VTOC 内の DSCB の内容を追加したり，変更したり，あるいは修正するような目的で同一 VTOC を同時にアクセスする必要が生じる．ましてや，2つ以上の CPU で，同一装置がアクセスされるようなシステムだと，余計に同一 VTOC 内の同一 DSCB が同時に利用される確率は高まる．したがって，同一システム内のタスク間や，シンテム間でのタスクが同時に同一ボリューム内の VTOC をアクセスする危険性を保護する必要性がでてくる．そこで，あるタスクが，ある VTOC を利用しているとき，その完全性を守るために，スペース管理ルーチンは，RESERVE，ENQ あるいは DEQ などのマクロ命令の機能を利用する．これらのマクロ命令は，その命令を出したタスクに対して，VTOC の利用を排他制御するようにしてくれるのである．これらマクロ命令の詳細については，第8章にて詳述するが，スペース管理ルーチンには，このような機能によって VTOC の完全性を保障することも要求されているのである．

八. データ制御ブロック (DCB) の機能と構成

1. DCB の基本構成と機能

データ・セットを適切に処理するためには，そのデータ・セットがどのボリュームに収容されているとか，どのような処理を要するとかの情報を記述しておく必要がある．この種のデータ・セットの属性をまとめて収容する記憶域を**データ制御ブロック**（Data Control Block：略して DCB）と呼ぶ．DCB は各々のデータ・セットに対応して必要であり，DCB マクロ命令を用いて処理プログラムの中に作り上げる．

DCB は，前述したように処理プログラムの一部を構成する記憶域であり，そのデータ・セットに関する諸々の情報について集約したものである．この DCB に要するスペースが，プログラムのアセンブルのさい，DCB マクロ命令によって確保されるのである．後述するように，アセンブルのさいには，必要なスペースが確保されるわけであって，データ・セットの特性を示すすべての

情報が集められるとは限らない．

アセンブラー言語以外の高級言語で書いたプログラムでも，各々のデータ・セットに対応する DCB が作られるが，この場合にはそれぞれの言語翻訳プログラムが DCB を作り上げる機能をもっている．したがって，高級言語ではプログラマーが直接 DCB を作成するわけではない．その代り，DD ステートメントの DCB 用パラメータを使って，DCB に必要な情報を与えることができる．もちろん，アセンブラー・プログラマーであっても，DD ステートメントで DCB の情報を与えることはできるが，あらかじめ原始プログラムの中で DCB を規定しておくことが必要なのである．

DCB マクロ命令を使って，直接 DCB を使り上げると同時に，その DCB の中に所要の情報を入れることもできる．また，プログラムの実行時に DCB のある種のフィールドの内容をダイナミックに変更することも許されている．あるプログラムで，そのプログラムをアセンブリーし直さないで，異ったデータ・セットをも処理することができるようにするためには，データ・セットをプログラムの中で参照するのにデータ・セット名を使わせないようにしておけばよい．つまり，特定データ・セットに直結するような名前は，実行時に与えるように工夫すればよい．そのために，データ・セット名だとかその他の関連情報（たとえば，装置の種類とかボリューム通し番号など）は，ジョブ制御ステートメントの１つである DD ステートメントで指示することができるようにしている．これによって，実行時に DCB のある種のフィールドの内容をダイナミックに変更できるのである．プログラマーは，プログラム作成時に各々のデータ・セットに関するわずらわしい指定から解放されるし，一方，完成したプログラムのデータ・セットに関する属性が変ったとしても，プログラムの書き直しをしなくてもすむのである，また，実行時にデータ・セットの属性の変更が可能なことから，さまざまな組合わせのテストを行い，最も処理効率のよいデータ・セットの組合わせの実現も可能となる．

さて，DCB に収容されるデータ・セットに関する情報であるが，実際にはつぎの種類に大別できる．

●データ・セット属性：レコードの長さ，レコードの形式，ブロックサイズあ

るいはデータ・セットの編成法などに関する情報．

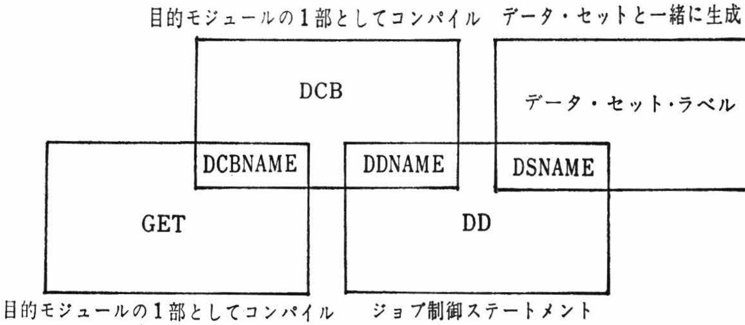
- 処理法の記述：バッファ確保の手法，処理の手法，装置に依存する情報，ある種のデータ編成法に適用される特殊な処理法など入出力処理にかかわりをもつ情報．
- 出口（エグジット）情報：入出力の特別な処理目的のために使用するルーチンのアドレスを示す情報．たとえば，全ての入出力レコードを読み終ったあとで特別な処理を行うためのルーチンであるとか，特別なラベル処理を行うためのルーチンなどのアドレスを貯えておく．

2. DCB の情報源

処理すべきデータ・セットに対応して，一対の DCB と DD ステートメントが必要なことはいうまでもない． DCB とそれに対応するデータ・セット間の仲介を行うのに DD ステートメントが機能する．処理プログラムの中では一切データ・セット名を使っていないが，DD ステートメントでは，対応する**データ・セットの名前**（DSNAME）を指定するからである．このデータ・セット名で該当するデータ・セットを探索することは以前にふれた通りである．ところで，DCB と DD ステートメントは，どのようにして関係づけを行うのであろうか．これに使われるのが，**データ定義名**（Data Definition Name：略して DDNAME）である． DDNAME は，DCB マクロの中の１つのパラメータとして定義される．一方， DD ステートメントにおいても， DDNAME が指定されるので，これによって両者の対応づけがなされる．

また，入出力マクロ命令では，そのオペランドで DCB の名前（DCBNAME）を指示することによって対応する DCB との関係づけを行う．このような記号による参照によって，**第28図**に図示したように DCB， DD ステートメントおよびデータ・セット・ラベル（ひいては，データ・セットそのものの）間の論理的結合が実現できるのである．

この論理的結合関係が判明するならば， DCB の各フィールドを満たすための情報源はどこから得られるかのおおよその見当はつくであろう． DCB マクロ命令，DD ステートメントおよびデータ・セット・ラベルの３つがそれであ



第28図　記号参照のチェイン

る．さらに，プログラムの実行時に，DCB の所要のフィールドに関連するデータを記憶させることによって，変更したり，追加したりすることもできる．

ともあれ，これら情報源にもとづいて入出力操作に必要な仕様が，OPEN マクロ命令による初期化手順を通じて決定されるのである．それゆえに，処理プログラムの書きかえという面倒な手続きを踏むことなしに，実行時にデータ・セットの仕様の選択が実現できる．

OPEN マクロ命令が実行されると，OPEN ルーチンはあらまし次の4つの機能を果してくれる．詳細は，本章第十節「データ・セットの OPEN と CLOSE 機能」にて取扱う．

- DCB を完全にする．
- 主記憶装置に必要なアクセス・ルーチンをロードする．
- データ・セット・ラベルや制御情報を読込んだり，書き出したりして，データ・セットの初期化をはかる．
- 必要な各種システム制御ブロックを作り上げる．

この4つの機能のうち第1の機能で DCB の各フィールドに必要な情報を貯えるのである（**第29図参照**）．

DD ステートメントから得られた情報は，ジョブ制御プログラムの機能によって**ジョブ・ファイル制御ブロック**（JFCB）に貯えられる．JFCB は，ジョブ・ステップで指定された各々の DD ステートメントに対応して作られ，いっ

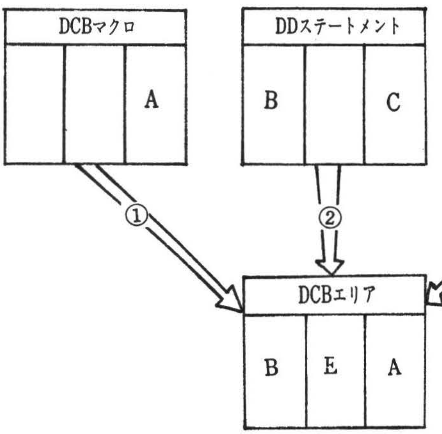
たん補助記憶装置に貯えられる．対応するDDNAMEをもったDCBがオープンされると，そのJFCBが主記憶装置に持込まれる仕組みになっている．

ともあれ，DCBは，DCBマクロ命令，JFCB（そのもとは，DDステートメントに変りないが）および，すでに存在するデータ・セット・ラベルからの情報を使って完成される．もしも，特定のフィールドに対して2つ以上の情報源がある場合でも，1つの情報だけが使われる．第29図で示しているように，DDステートメントの方がデータ・セット・ラベルに優先する．DCBマクロ命令の情報は，更に両者よりも優先的に使われる．したがって，DCBマクロ命令では必要最少限ないしは極めて固定的内容のフィールドだけの指定を行っておいた方が，融通性に富んだものとなる．なお，DCBの完成する様子を図解したのが**第30図**である．

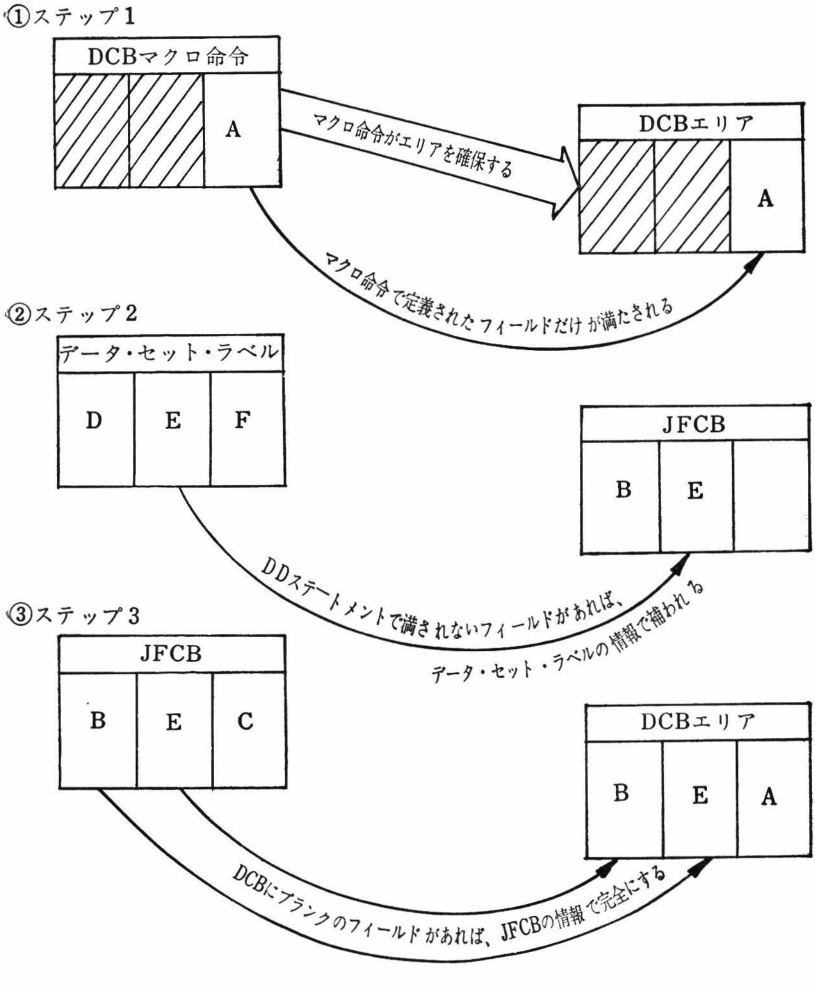
このようにして完成したDCBの中の各フィールドの内容をDCBエグジット・ルーチンを使って変更することもできる．このルーチンは，ユーザが作成するものであり，意図する内容で条件に応じた変更が許される．

3. DCBとDDステートメント

DCBを構成するフィールドに貯えられる情報は，大別して，データ・セッ



第29図　データ制御ブロックの生成



第30図　OPEN マクロ命令によるデータ制御ブロック（DCB）の完成

ト属性，処理法の記述および出口（エグジット）情報の3種類になることはすでにふれたが，ここで，その主要なフィールドについてまとめておこう．以下に述べる DCB のフィールドは，ほとんどのデータ編成法やアクセス手法に共通する代表的フィールドだけである．

1) データ・セット属性

データ・セット属性の代表的なものとしては，データ・セットの編成法，レコードの形式，レコードの長さ，ブロックサイズなどがある．

データ・セットの編成法 (DSORG) のパラメータでは，順次編成 (PS)，索引順次編成 (IS)，区分編成 (PO)，直接編成 (DA) などの編成法の指定ができる．このパラメータは，DCB マクロ命令の中で必ず指定する必要のあるものの1つである．索引順次編成と直接編成のデータ・セットの場合には，DD ステートメントの中でもこの情報を指定する必要がある．

レコード形式 (RECFM) パラメータでは，データ・セットを構成するレコードの特性を指定する．固定長 (F)，可変長 (V)，不定形式 (U) などがそれである．レコードがブロック化されている場合には，FB，VB などと指定する．

レコードの長さ (LRECL) パラメータでは，データ・セット内の各レコードの長さをバイト数で指定する．レコードの形式が可変長である場合には，例によって最大のレコードの長さを指定することになる．不定形式のレコードに対しては，このフィールドの指定は不用である．

ブロックサイズ (BLKSIZE) のパラメータでは，ブロックの最大の長さをバイト数で指定する．レコード形式が固定長であるなら，ブロックサイズは，レコードの長さの整数倍である．レコード形式が可変長なら，最大のブロックサイズを指定することになる．一方，不定形式のレコードの場合だと，レコードの長さよりも4バイドだけ大きい長さを指定する．

以上の他に，データ・セットの属性としては，テープの記録密度，キーの長さ，印刷装置の行間隔などさまざまなフィールドがある．

2) 処理法の記述

入出力操作を適切にコントロールさせるためには，制御プログラムに伝達しておくべき幾種類かの処理情報が必要である．処理プログラムの中で用いている入出力関係のマクロ命令の形式，入出力バッファに関する情報，あるいは特別な処理ルーチンのアドレスなどがこれに当る．

マクロ命令は，プログラムがアセンブルされるさいにそれに見合ったルーチ

ンが作り出されるゆえに，データ・セットを処理するために使う入出力マクロの形式を DCB マクロ命令の中であらかじめ定義しておくことが要求される．

バッファの要件だとかそれに関連する情報は，DCB マクロ命令，DD ステートメントあるいはユーザ・ルーチンで DCB の対応するフィールドに必要な情報を貯える方法でみたすことができよう．

特別な処理ルーチンのアドレスをマクロ命令で指定しない場合には，そのデータ・セットをオープンする前にそのルーチンのアドレスを該当フィールドに入れる必要が生じる．

3) 出口（エグジット）情報

出口情報とは，入出力処理に関する特別な状態や条件が生じたときにブランチすべきユーザ・ルーチンのアドレスを示す情報のことである．たとえば，データ・セットの終りに達したとき（End of Data Set）とか，**入出力に関するエラー**（synchronous error）が生じたとき，ユーザ・ラベルの処理の必要性が生じたとき，あるいは，ボリュームの終りに達したときなどが，特別な状態や条件に当る．これら状態や条件が生じたときにブランチすべきユーザ・ルーチンのアドレスが貯えられるフィールドが DCB の中にもうけられるのであるる．

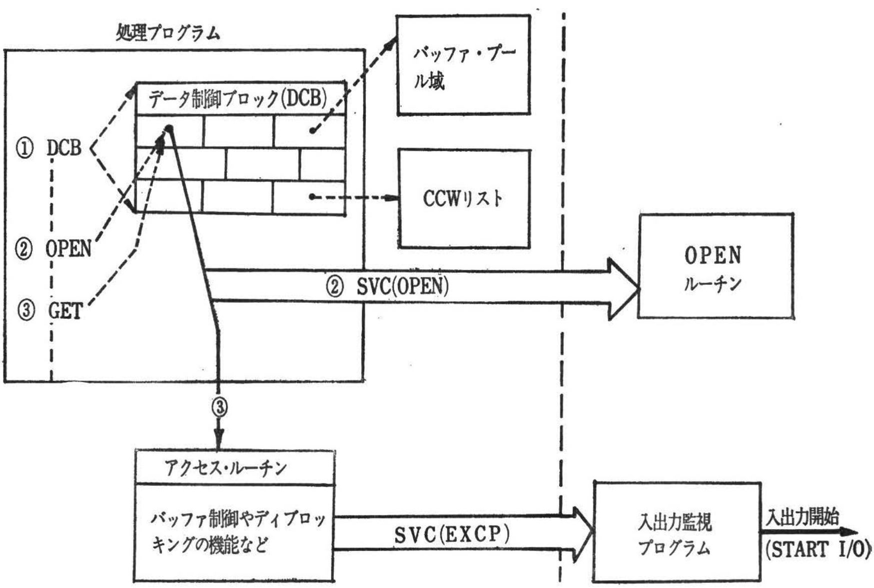
以上に示したような DCB フィールドに必要な情報は，DCB マクロ命令を通して与えることができるのは当然であるが，DD ステートメントを使って与えることのできるものが多い．特にデータ・セット属性のほとんどのものは，このステートメントで指定することが可能である．

DD ステートメントを使うことによって，プログラムの作成時ではなく，そのプログラムが実行される時点にバッファの大きさだとか，レコードの長さあるいは装置のタイプなどの諸属性を選択指示できるのである．この DD ステートメントは，単にデータ・セットの属性を指示するだけでなく，データ・セットの識別名や装置に関する情報，データ・セット・ラベルに関する情報あるいはデータ・セットの後始末に関する情報などの指示をも行う．とはいうもののプログラムの実行時にデータ・セットの諸属性が指示できるのは，DD ステー

トメントの主要な特徴の１つといってもよかろう．DD ステートメントのオペランド（DCB パラメータ）を使ってデータ・セットの属性を指定する．DCB マクロ命令のオペランドで指定することのできる大半のパラメータが，この DD ステートメントの DCB パラメータでも指定できるわけである．なお，DD ステートメントの詳細については，第７章「ジョブ管理の構造と手法」にゆずることにする．

これまで説明してきた DCB の機能について構造的にまとめることにしようう．その機能図を示したのが**第31図**である．

DCB が必要とする記憶域は，DCB マクロ命令によって確保される．ただしアセンブラー言語以外では，言語翻訳プログラムの責任で，このスペースを自動的に確保してくれる．DCB を構成するフィールドに貯えられる情報は，データ・セット属性，処理法の記述，出口の情報などである．データ・セット属性では，属性の情報そのものが該当フィールドに貯えられるのであるが，出口の情報や処理法の記述では，単にアドレスだけが貯えられる．このアドレスを



第31図　入出力プログラムの流れ

参照することによって，しかるべき処理ルーチンにブランチするのである．

ともあれ，プログラムが翻訳された時点では，DCB の各フィールドは完成していない．一部のフィールドは，DCB マクロ命令で指定されたパラメータによってうめられているが，他のフィールドはそのデータ・セットがオープンされる時点に適切な情報でうめられる．DCB マクロ命令以外の情報源は，JFCB とデータ・セット・ラベルである．JFCB は，DD ステートメントの情報を集約して補助記憶装置に一時点に貯えているブロックのことである．データ・セットがオープンされることにより主記憶装置に持込まれ，DCB の空きのフィールドを該当する情報で補う．DD ステートメントは実行時に指定できることから，目的に応じて，データ・セットの属性を変えることができるという便利さがある．JFCB をしてもなおかつ空きのフィールドがある場合には，該当データ・セットのラベル上に収容されている情報で空きフィールドをうめて完全な DCB とする．必要なら，DCB の出口ルーチンを使って，ユーザ・プログラムでフィールドの内容を追加あるいは変更することもできる．

OPEN ルーチンは，このようにして DCB を完全にするだけでなく，そのデータ・セットが利用するアクセス・ルーチンのアドレス，チャネル・プログラム（CCW リスト）のアドレスあるいは確保されたバッファ・プール域のアドレスなどをも，DCB の所定フィールドに貯えて利用可能な DCB にしてくれる．

このようにして出来上った DCB は，入出力マクロ命令が与えられる都度，あるいはアクセス・ルーチンが実行される間中絶えず参照される．これにより適切な入出力処理が達成されるのである．DCB はデータ・セットに関するあらゆる情報が集約された，入出力操作において欠くことのできない中枢的，中継的機能を果す制御ブロックである．

九．バッファの確保と制御機能

入出力オペレーションの重要な要素として，入力後のデータおよび出力前のデータを一時的に保持しておく区域を主記憶装置内に確保する仕事がある．デ

ータ管理プログラムは，主記憶装置をバッファとして取得したり，取得したバッファに入力レコードを取り入れるためのスケジュールをしたり，バッファ内のデータをプログラムで処理できるようにするための諸機能を備えている．

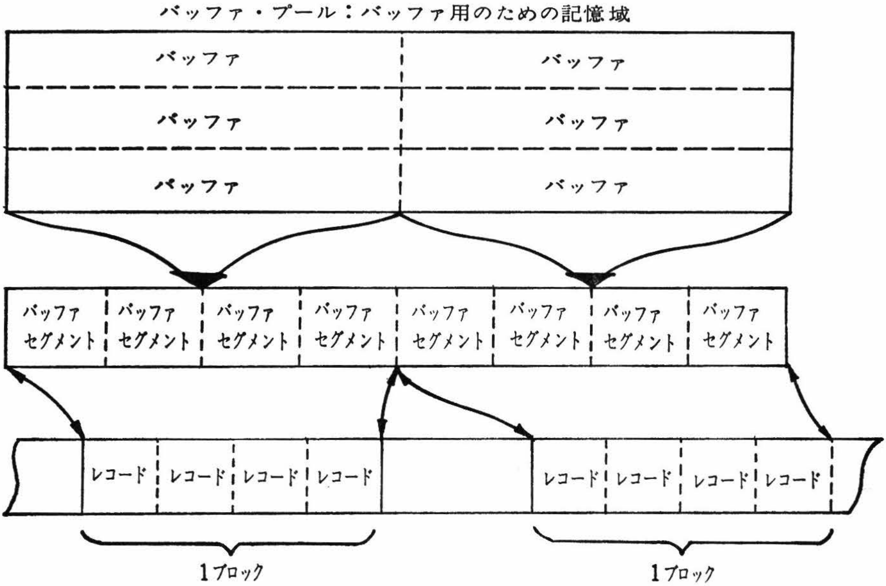
ところで，ここでいう**バッファ**とは，主記憶装置と外部記憶装置（入出力装置）間で転送するデータ・セット・レコードを一時的に貯える主記憶装置の区域のことである．**データ・セット・レコード**というのは，論理レコードではなく，外部媒体に物理的に記憶されるレコードの単位，つまりレコード・ブロックのことをいう．そこで，バッファの中で，論理レコード１つを収容する部分を特に**バッファ・セグメント**（buffer segment）と呼ぶことによって，１ブロック分を貯えるバッファと区別する．また，主記憶装置の中で，データ管理プログラムが制御情報にもとづいて，形式を定めてくれた区域に入っている一群のバッファのことを**バッファ・プール**（buffer pool）という．これら３つの記憶域の相互関係を図示したのが**第32図**である．

このようなバッファに関する管理機能を検討すると３つの側面があることに気がつく．まず第１は，データ転送に必要となるバッファを取得するために設けられるバッファ・プールをどのようにして設定すべきかの問題である．プログラマーに完全に任かせるのがよいか，またはシステムが全面的にその面倒をみるのがよいか．

ブロックサイズやバッファの数が最初から固定的であれば，プログラマーに任せるのが得策であるといえるが，実行時にそれらが決定されるような場合だとシステムにプールの設定を委ねた方がよい．ともあれ，バッファ・プールの設定に関する問題がまず生じる．

つぎには，そのプールからどのようにして特定のバッファを選択するのが効果的ないしは効率的かという問題に直面する．いくつかのバッファのうちから空きのバッファを選び出すという仕事をプログラマーに行わせることは負担が多すぎる．バッファの割当ては，できるだけシステムに任かせた方がよかろう．

バッファ・プールが設定され，各入出力ごとのバッファの割当てが定まるとつぎはどのような転送方法が望ましいのかという問題が生じる．バッファの中でそのまま処理するような方法がよいのか，それとも１たん作業域に移動した



第32図 バッファ・プール，バッファおよびバッファ・セグメントの関係

うえで処理した仕方がよいのかということになる．さらには，コンピュータ・ハードウェアの入出力機能である**分散読取り**(scatter read)や**集合書出し**(gather write)を生かした転送方法を選ぶべきかということにも発展する．

このような側面を考えてみると，いずれの場合にも使用意図と目的に応じた選択ができるような融通性に富んだ機能が提供されることこそが要求される．そこで，データ管理プログラムは，それぞれの側面に関して選択可能ないくつかの方法を提供してくれる．

まず，**バッファ・プールの設定**であるが，これには3つの方法がある．第1の方法は，プログラマーがGETPOOLマクロ命令を使うことによって，プログラムの実行時にバッファ・プールに対するエリアを主記憶装置内に割り当て，バッファ・プールを設定してくれるようにシステムに要求する方法である．GETPOOLマクロ命令のオペランドでは，該当DCBのアドレス，バッファの長さ，および必要なバッファの個数を指定する必要がある．設定したプ

ールがもはや必要なくなった時点では，FREEPOOL マクロ命令によってその記憶域を解放することができる.

データ管理プログラムにバッファ・プールの設定を委ねるよりも効果的な主記憶域の割当てができることをプログラマーが知っておれば，バッファ・プールのエリアを自分で確保してしまうこともできる．この方法だと，たとえば，もはや必要でなくなったサブルーチンなどで占めている記憶域をバッファ・プールとすることもできる．プログラマーは，BUILD マクロ命令を使って，自分が確保した場所をバッファ・プールとして設定するようにシステムに要求を出す必要がある.

第3の方法は，OPENルーチンに自動的に主記憶装置を確保させ，プログラマーに代って，バッファ・プールを設定させる方法である.

さて，バッファの割当てに先立って，レコードの転送方式について考えてみよう．たとえば，磁気テープのレコードを処理する場合はどうであろう．1つあるいは2つ以上のデータ・セットからレコードを読んで処理し，1つあるいは2つ以上の新しいデータ・セットを作るという仕事が普通である．この場合種々のバッファ使用法が考えられる．レコードをバッファ内でそのまま処理するだけで十分な場合もあろう．入力レコードを作業域に移して処理し，更新されたレコードを作業域から出力バッファに移すという方法が望ましい場合もあろう．その他の手法も場合に応じて用いられよう.

直接アクセス記憶装置のレコードだと，同じデータ・セットを入力と出力に使用して，更新作業を行うことがしばしばある.

このようなことから，レコード転送ルーチンでは，3種類の転送モード，つまり移動方式，位置指定方式，置換方式を取扱うことができるようになっている．**移動方式**(move mode) だと，各論理レコードは入力バッファから作業域に移され，ここで処理が行われたあと出力バッファに移動される．**位置指定方式** (locate mode) では，レコードの移動は行わず，レコードの入るバッファ・セグメントの最初を示すポインター（レコードの最初のアドレスを指すもの）をプログラマーが使用できるという方法である．一方，**置換方式** (substitute mode) では，やはりポインターが用いられ，適用業務プログラムでは，論理レ

コードのバイト数に等しい作業域を設け，バッファ・セグメントと作業域が効率よくその役割を交換し合う．

転送モードを補足するために3つのバッファ割当て方法がある．単純バッファ割当法，交換バッファ割当法および連鎖セグメント・バッファ割当法の機能がある．**単純バッファ割当法**（simple buffering）は，最も一般的であり，1つあるいは2つ以上のバッファを各データ・セットに割り当てる．

交換バッファ割当法（exchange buffering）は，固定長レコードだけに用いられ，“分散読取り”と“集合書出し”のオペレーションを行い易くするためのデータ・チェイングの機能を活用している．入力データ・セットのためのバッファ・セグメントは，出力データ・セットのバッファ・セグメント，作業域のバッファ・セグメントと自動的にその役割が交換される．各々のバッファ・セグメントが順番に入力域，作業域，および出力域として取り扱われるだけでなく，互に連続していないセグメントがあたかも1つのブロックとして取扱えるのである．これこそ，データ・チェイニングの機能である．

連鎖セグメント・バッファ割当法（chained segment buffering）は，可変長のメッセージを取扱うための手法である．物理的には分離している複数個のセグメントの順序づけをチェイニングの手法を利用して行い，セグメントが動的に形式されてゆく．この手法は遠隔地の端末装置との間での可変長メッセージのやりとりをするための記憶域の割当てを行うという必要性から解放してくれるものである．

データ管理プログラムの管理機能について概観してきたが，以下でバッファ・プールの設定，バッファ割当手法，データの転送方式についてより具体的に説明してゆこう．

1.　バッファ・プールの設定手法

OPENされる**データ制御ブロック**（DCB）のそれぞれに対応してバッファ・プールを設定することが必要である．しかも，プール内のスペースがデータ転送のためのバッファとして割当てられるまえに，所要のバッファ・サイズとその数を示す情報をバッファ制御のためにプールの一部にあらかじめ貯えておく

必要がある．

実際には，この制御情報は，バッファ・プールの最初の8バイトに貯えられるようになっている．そして，この8バイトが**バッファ制御ブロック**（Buffer Control Block：DCB では BUFCB パラメータでこのエリアを指示する）として参照される．

バッファ・プールを確保し，バッファの大きさと個数に関する情報をバッファ制御ブロックに記憶するプロセスが，バッファ・プールの設定である．前述したごとくバッファ・プールの設定方法には3種類がある． OPEN マクロ命令でシステムに自動的に行わせる方法，プログラマーがプールに必要な区域をあらかじめ確保しておき， BUILD マクロ命令でバッファ・プールに設定する方法，および GETPOOL マクロ命令を使って実行時にプールに必要な記憶域を確保しバッファ・プールを設定する方法である．この3つの方法について具体的に説明しよう．

1） OPEN マクロ命令による自動設定機能

プログラマーが GETPOOL マクロ命令や BUILD マクロ命令を使ってバッファ・プール設定手法を指示していない場合には，OPENルーチンが自動的にバッファ・プールを確保してくれる．

DCB が完成されると，OPEN ルーチンはつぎの3つのフィールドの情報を使ってバッファ・プールの設定を行うのである．

● BUFNO：プールの中に確保すべきバッファの数

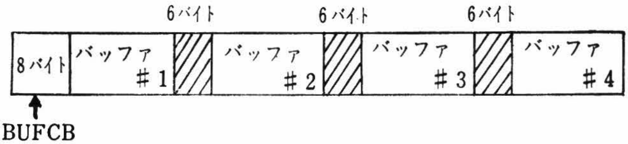
● BUFL：各バッファの長さ，つまりバイト数

● BFALN：各バッファの境界合わせ，つまりバッファの最初のバイトが倍ワード（Dを指定），ワード（Fを指定）境界から始まる必要があるかの指示．

この3つの情報を使って，バッファ・プールを設定してくれるのである．バッファ・プールの自動設定の例を示したのが**第33図**である．

この図でお分りのように，プログラマーは DCB マクロ命令の中で，あるいはDD ステートメントを通じて BUFNO，BUFL，BFALN の指示をする必要がある．なお，EODAD パラメーターでは，すべての入力レコードを処理した

```
           :                                     処理
           :
         OPEN       (INDCB)
           :
           :                                     処理
OWARI    CLOSE      (INDCB)      すべての入出力が終ったあとバッファ・プールを
         FREEPOOL   INDCB                                            解放する
           :
           :
         RETURN                  制御プログラムにコントロールを返す
INDCB    DCB        BUFNO=4, BUFL=50, BFALN=D, EODAD=OWARI,……
```



第33図　OPENマクロ命令によるバッファ・プールの自動設定

あとブランチすべきルーチンを指定する．

この例では，バッファの境界合わせを倍ワード（D）としているので，ちょうど各バッファが倍ワードから始まるように自動的に調整を行ってくれる．その結果，6バイトの空きが生じるのである．また，バッファ・プールの最初には8バイトのバッファ制御ブロックが作られ，バッファのサイズと個数の情報が入れられる．さらに，OPENルーチンは，このブロックのアドレスをDCBのBUFCBフィールドに収容する．こうすることによって，DCBを参照すれば，バッファ制御ブロックやバッファ・プールの所在をつきとめられる．

バッファ・プールの設定にOPENルーチンを使うやり方は，最も便利な方法である．しかし，この方法を使うにさいしては，確保されたバッファ・プールがただ1つのDCBにしか割当てられないということを考慮しておく必要がある．後述するBUILDマクロ命令の方法だと，1つのプールを2つ以上のDCBに割当てることができるので，それだけ主記憶域を有効に使えるわけである．

ともあれ，OPENルーチンでプール区域を取得するのには，GETMAINと

いう命令（これは，主記憶装置を実行時に動的に確保する命令であり，第8章「タスク管理機能と多重プログラミング」にて詳述予定）を監視プログラムに出すやり方をとる．この要求によって主記憶装置が取得されプールに割当てられるわけであるが，その後，そのタスクの終了時には FREEMAIN 命令が出されることによって，確保されていた記憶域が自動的に監視プログラムに空いた区域として返上されるのである．タスクの終了時ではなく，そのデータ・セットに関する入出力オペレーションが完全に終了した時点にプールを解放するには，**第33図**に示したように CLOSE マクロを与えた直後に FREEPOOL マクロ命令を出せばよい．この命令によって，FREEMAIN と同じ機能が果され，プールとしての主記憶域は空きのエリアとしてシステムに返されるのである．返上された空きの区域は，別の命令によって主記憶装置が要求されたときに役立てられる．

2) BUILD マクロ命令によるバッファ・プールの設定

BUILD マクロ命令によるバッファ・プールの設定の仕方は，**第34図**に示しているように，つぎの手順に従う．

プログラマーは，問題処理プログラムの中で DS ステートメントを使って，バッファ・プールに必要なエリアを定義する．DS ステートメントを用意するに当っては，要求するバッファの個数，各々のバッファのサイズおよび境界合わせに要するスペースなどを考慮する必要がある．さらに，バッファ制御ブロックとして必要な8バイトを付加したエリアとしなくてはならない．**第34図**では，INDCB と OUTDCB と名付けた2つの DCB にそれぞれ50バイトのバッファを5個用意するために500バイトが必要であり，さらに8バイトがバッファ制御ブロックとして必要であるので，合計508バイトのエリアを定義する．この例では，2つのデータ・セットが1つのバッファ・プールを共有している．というのは，どちらも BUFCB で INPOOL を指定しているからである．

このようにしてアセンブリー時にプールとして必要な記憶域を確保したうえで，実行時には，その区域をバッファ・プールとして設定するための BUILD マクロ命令を出す．この命令が実行されると，バッファ制御ブロック，つま

```
         :                                  処理
         BUILD   INPOOL, 10, 50             50バイトのバッファを10個もった
                                            バッファ・プールを設定.
         OPEN    (INDCB,, OUTDCB, (OUTPUT))
         :                                  処理
ENDJOB   CLOSE   (INDCB,, OUTDCB)
         :                                  処理
         RETURN                             制御プログラムにコントロールを
                                            返す
INDCB    DCB     BUFNO=5, BUFCB=INPOOL, EODAD=ENDJOB, …
OUTDCB   DCB     BUFNO=5, BUFCB=INPOOL, …
         DS      0D                         倍ワードに境界合せを行う
INPOOL   DS      508                        バッファ・プールとして508バイ
                                            ト確保
```

第34図　BUILD マクロ命令によるバッファ・プールの静的設定

り，プログラムの中で定義したエリアの最初8バイトにバッファの個数やサイズなどのコントロール情報が貯えられるのである．第34図では，50バイトのバッファを10個設定するように要求している．なお，BUILD 命令は，OPEN 命令より前に実行させる必要がある．

OPEN ルーチンでも，この方法と実質的には同じ機能を自動的に行ってくれるわけであるが，第34図の例でもおわかりの通り，1つのプールを2つ以上の DCB に割当てることができるので，それだけ融通性に富んだ主記憶域の利用ができるのである．さらに，データ・セットを CLOSE したあとで，同じエリアを別の目的に再使用することもできるし，再び BUILD マクロ命令を出して別のデータ・セットが用いる新しいバッファ・プールに作り直すこともできるという利点もある．

3) GETPOOL マクロ命令によるバッファ・プールの設定

3番目のプールを設定するやり方では，GETPOOL マクロ命令を使用する．GETPOOL マクロ命令も，OPEN ルーチンでプールを作るのと同じ機能を果す．ただし，GETPOOL マクロ命令は，それが関係する DCB を OPEN するまえ，または，DCB の出口ルーチンの中で実行させる必要がある．

第35図で示しているように GETPOOL マクロ命令のオペランドでは，DCB のアドレス，プールに必要なバッファの個数，各バッファを構成するバイト数を指示する必要がある．

GETPOOL で確保したエリアが使用済になったら， OPEN の場合と同様 FREEPOOL マクロ命令を出してその区域をシステムに返上することができ

```
                :                              処理
                :
                BE          POOLB              } どのようなバッファ・プールを設定す
                BL          POOLC              } るかをテスト
                GETPOOL     INDCB, 3, 100      100 バイトのバッファを3個もったバッ
                                               ファ・プールを設定
                B           OPENRTN
POOLB           GETPOOL     INDCB, 5, 500      500 バイトのバッファを5個もったバッ
                                               ファ・プールを設定
                B           OPENRTN
POOLC           GETPOOL     INDCB, 5, 100      100 バイトのバッファを5個もったバッ
                                               ファ・プールを設定
OPENRTN         OPEN        (INDCB)
                :                              処理
                :
ENDJOB          CLOSE       (INDCB)
                FREEPOOL    INDCB              すべての入出力が終ったあとバッファ・
                                               プールを解放する
                :                              処理
                RETURN
INDCB           DCB    DSORG=PS, BFALN=F, RECFM=F, EODAD=ENDJOB, …
```

第35図 GETPOOL マクロ命令によるバッファ・プールの明示的設定

る．この命令を使わないときは，タスクの終了時にそのスペースが解放される．

GETPOOL による方法も，OPEN ルーチンの場合と同じ機能を果すわけであるが，実行時にバッファの数やサイズを条件に応じて変更したいような場合には都合のよいやり方である．**第35図**に示したようにどのようなバッファ・プールにすべきかの条件をテストしたうえで，その条件に見合った GETPOOL マクロ命令を実行させるようにすればよいからである．

2. バッファの割当て手法

バッファ・プールの中から特定のバッファを選び出すやり方は，使用するアクセス方式，つまり，**待機アクセス方式**（queued access method）か**基本アクセス方式**（basic access method）かによって異る．したがって，バッファ・プールからのバッファの選択割当てに関しては，２つのアクセス手法に分けて言及しよう．

1) 基本アクセス方式の場合の割当て方法

基本アクセス方式では，プールからのバッファ・エリアの選択をプログラマーがコントロールする方法とシステムが動的に割当てる方法がある．

第１の方法はプログラマーが，入出力マクロ命令を実行するまえにGETBUFマクロ命令によって，プールからバッファ・エリアを確保する．入出力処理が終ると FREEBUF マクロ命令で使用ずみのバッファをプールに返す方法をとる．場合によっては，プール以外の記憶域をバッファとして利用してもよい．いずれにしろ，プログラマー自身が READ および WRITE マクロ命令で使用するバッファを指示することが要求される．**第36図**で GETBUF と FREEBUF を使ったバッファ制御の例を示している．GETBUF のオペランドで DCB の名前とプールから選択されたバッファのアドレスを入れるレジスターを指定する．READ や WRITE マクロ命令のオペランドでは，割当てられたバッファ・アドレスの入ったレジスターを指示することによって，読み書きに使う記憶域を知らせるのである．入出力処理がすむと，バッファをプールに返すために FREEBUF を使う．オペランドでは GETBUF と同じように DCB の名前とバッファ・アドレスの入ったレジスターを指定する．

基本アクセス方式のうちの BISAM と BDAM では，制御プログラムが自動的にバッファ・エリアを割当てゝくれる．ただし，DCB の所定のフィールドで**動的バッファ割当て**（dynamic buffering）を要求しておく必要がある．こうすることによって，データ転送の直前にシステムがバッファ・エリアを割当ててくれるので，プログラマーがプールからバッファ・エリアを取得するた

```
         :
         OPEN      (INDCB)
         :
         GETBUF    INDCB, ( 5 )            プールからバッファを確保（レジスタ
         :                                 ー5に確保されたバッファの最初のア
         READ      —, —, INDCB, ( 5 )      ドレスが入る）
         :
         FREEBUF   INDCB, ( 5 )
         :
OWARI    CLOSE     (INDCB)
         FREEPOOL  INDCB
         :
         RETURN
INDCB    DCB       BUFNO= 4 , BUFL=50, BFALN=D, EODAD=OWARI, …
```

第36図 基本アクセス方式（READ／WRITE マクロ命令）の場合の
バッファの制御

めの要求命令を出す必要はなくなる．割当てられたバッファは，その内容が書き出されたとき自動的にプールに戻されるか，プログラマーがマクロ命令で戻す．バッファはプールに返されたのち，別の READ 要求のために動的に割当てられる．この手法だと主記憶装置の利用効率が高まるので，出来るだけ活用すべきである．

2） 待機アクセス方式の場合の割当て方法

待機アクセス方式では，バッファ割当てに関するあらゆる制御が自動的に行われ，入力，処理および出力が最大限オーバラップするようにしてくれる．

プログラマーが GET および PUT マクロ命令を出すと，プール内のどのバッファ・エリアを入力，処理あるいは出力に割当てるべきかを自動的に決定してくれる．したがって，プログラマー自身がバッファ割当てに関与する必要はない．

しかし，レコード処理にさいしてバッファを有効利用するためにはいくつかの考慮事項がある．これは，1たん割当てられたバッファ内のレコードがどの

程度頻繁に移動するかにかかわりをもつ問題である．

入力レコードの場合だと，バッファから作業域に移したあとで処理する方法とバッファの中でそのまま処理する方法が考えられる．

一方，入力レコードが出力レコードとして利用されるような場合だとつぎの3つの方法が考えられる．

- 入力バッファから作業域に移して処理し，そのあとで出力域に移す．
- 入力バッファで処理したうえで，出力バッファに移す．
- 入力バッファで処理し，そのバッファ・エリアからそのまま出力する．

バッファのデータ移動に影響をおよぼすこれら要素は，つぎの項目で取扱う，「転送方式とバッファ割当て手法」で説明する．

3.　転送方式とバッファ割当て手法

転送方式とは，どのような方法で入力バッファのデータをユーザー・プログラムで取扱えるようにするか，あるいは，どのような方法で出力用データをバッファに入れるかにかかわりをもつ問題である．このような機能を果すためにデータ管理プログラムではつぎに示す3つの転送方式を用意している．

- **移動方式**（move mode）：GET マクロ命令が実行されると入力バッファから作業域にデータが移動する．逆に PUT マクロ命令だと，作業域から出力バッファにデータ移動する．
- **位置指定方式**（locate mode）：GET にしろ PUT にしろデータの移動はしない．しかし，バッファ内の所要レコードの位置をポインターで示してくれる．ポインターとは，処理の対処となるレコードの入っているバッファ・セグメントのアドレスを特定レジスターで指示してくれる方法である．
- **置換方式**（substitute mode）：位置指定方式とよく似かよっているが，プログラマーがバッファ・セグメントに等しい作業域を用意する必要がある点で異る．この作業域がバッファ・セグメントに置き換わる．つまり，入出力バッファ・セグメントと用意した作業域が必要に応じてその役割を交替する方式である．

このようなデータ転送方式に関連して，バッファ割当て手法にはいくつかの

ものが存在する．**バッファ割当て手法**（buffering technique）とは，バッファ・エリアをプールの中から自動的に選定して割当ててくれる手法であり，データ管理プログラムの行ってくれる機能であることはいうまでもない．自動的バッファ割当て手法の代表的なものとして，ここでは**単純バッファ割当て**（simple buffering）と**交換バッファ割当て**（exchange buffering）についてふれる．

1） 単純バッファ割当て

この方法は，もっとも融通性に富んだ，かつ適用範囲の広いものである．この方法では，各データ・セットに対して１つ以上のバッファが使われる．それぞれのバッファは，最大の長さのブロックを保つに十分な大きさのものである．単純バッファ割当てにおいては，バッファは割当てられたデータ・セットにだけ使用される．入力データ・セットに割当てられたバッファは，出力データ・セットでは使用できない．この点は，後述する交換バッファ割当てとは著るしく異る．

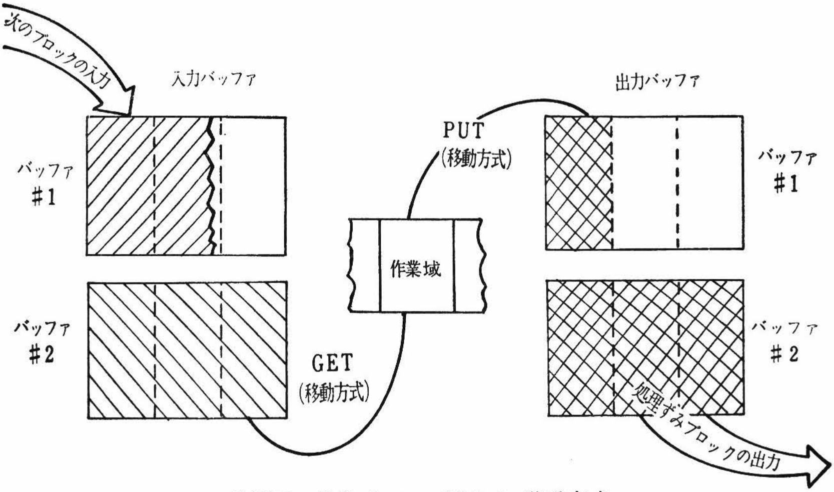
単純バッファ割当てでは，データ転送方式に関連してつぎのいずれかの方法で論理レコードが取扱われる．

●入出力ともに移動方式（**第 37 図参照**）：GET 命令によって入力バッファの所要レコードは作業域に移される．したがって，データの処理は作業域で行なう． GET 命令とは無関係に空きの入力バッファには，つぎのブロックの入力が併行して行われる．

一方， PUT 命令によって，作業域の論理レコードは出力バッファの所定の位置に移される．論理レコードで満された出力バッファのブロックは，入力や処理と併行して書き出される．

●位置指定方式の入力と移動方式の出力（**第 38 図参照**）：GET 命令は位置指定方式なので，この命令が与えられることにより，所要レコードの入っているバッファ・セグメントのアドレスがレジスターに返される．このアドレスを使用して，その場でデータの処理が行なわれる．

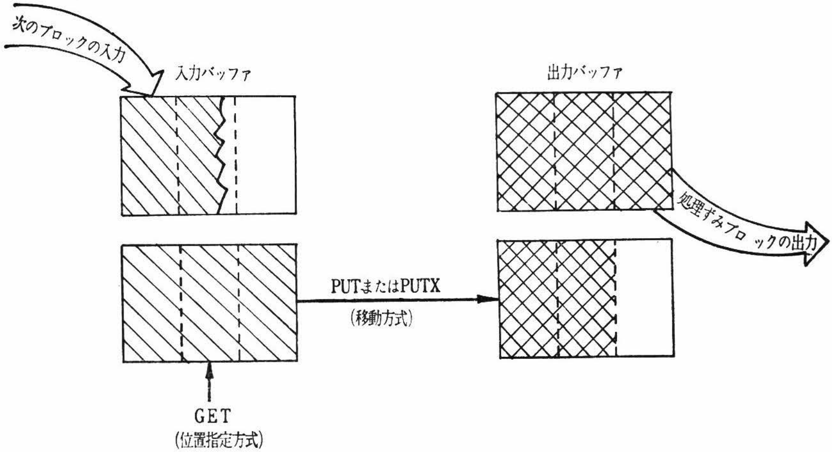
一方， PUT または PUTX 命令は移動方式であるので，入力バッファの



第37図　単純バッファ割当て—移動方式

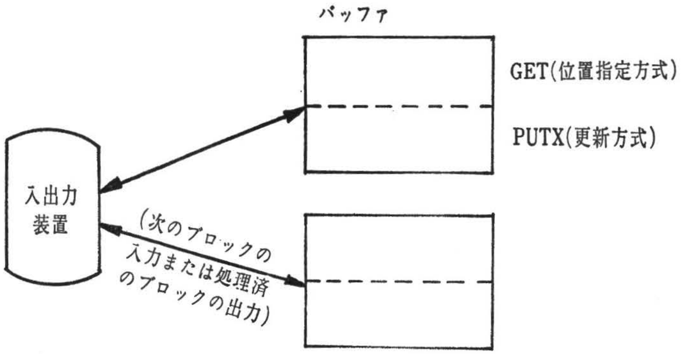
所定セグメントに入っている論理レコードが出力バッファの所定セグメントに移される.

入力，処理，出力の併行操作は，「入出力ともに移動方式」の場合と同じである.



第38図　単純バッファ割当て—位置指定方式の入力と移動方式の出力

●位置指定方式の入力と更新（**第39図参照**）：この方式は，直接アクセス記憶装置にあるデータ・セットのレコードを入力して更新処理を行ない，もとの



第39図　単純バッファ割当て―その場で更新

場所に書きもどすときに適用される．

位置指定方式の GET で，対象となる論理レコードのアドレスを知ったうえで，レコードの更新作業を行う。そのあとで更新方式の PUTX 命令を与える．

レコードの更新処理に併行して，別のブロックの入力あるいは更新ずみブロックの書きもどしが行われる．

2)　交換バッファの割当て

交換バッファ割当ての手法は，置換方式のデータ転送を実現する方法でありその基本的ねらいは主記憶装置内で論理レコードの移動を伴わないで処理が行えるようなバッファ・エリアの管理を行うことにある．

コントロールの仕方は，バッファ・エリア単位というよりもバッファ・セグメント単位で行われる．

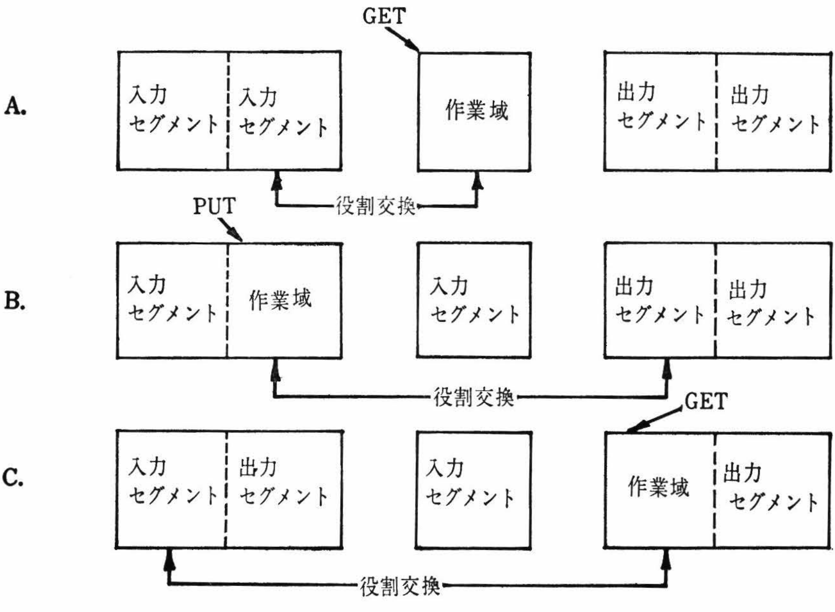
第40図で図示しているように論理レコードは，あるセグメントから別のセグメントに移動することはない．

その代り，それぞれのセグメントの機能を，入力域，作業域および出力域という具合いに交換して取扱う．入力レコードの入っているセグメントは，GET

命令により作業域となり，その場で処理が行える．代りに GET オペランド命令で与えた作業域は，その役割りを入力域に変えることによって，入力セグメントの不足を補う．処理した作業域を PUT 命令で指示することによって，作業域は出力域に変る一方，出力セグメントの中の１つが作業域に役割りを変える．この作業域をつぎの GET 命令で指示することによって，作業域は入力セグメントとなる．このようにして，同一のセグメントが，入力セグメント，作業域，出力セグメントとしてその役割を順次交換してゆくのである．この方法だと，論理レコードは一切移動しなくてすむ．

このように連続しないセグメントを１つのブロックとして取扱うのに，チャンネル・プログラムの中で**データ連鎖**(data chaining) の機能を使っているのである．この方法によって，分散読取りや集合書出しの操作が可能となっている．

交換バッファ割当ての手法は，２つ以上のデータ・セットを組合わせるような仕事や論理レコードの追加・削除を伴うような処理において有効であり，か



第40図　交換バッファ割当て

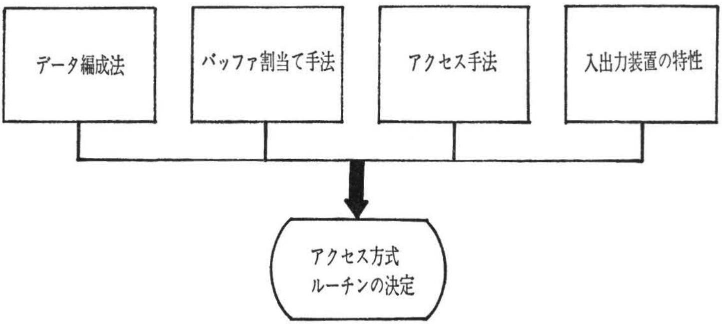
つ効果的である.

以上, バッファ・プールの設定とバッファ制御について, アセンブラー言語を用いた場合を主体にして説明してきたが, COBOL や FORTRAN などの高級言語においてもそのやり方に大差はない. ただし, このような言語レベルでは, バッファ・プールやバッファ制御の特別な命令は用意されていないので, 原則的には自動的バッファ・プールの設定機能やバッファの割当て機能しか使われていない. また, 自動的バッファ割当てでは, 単純割当てしか利用していないのが実情である.

十. データ・セットの OPEN と CLOSE 機能

プログラムが目的プログラムに翻訳された段階では, まだ, 入出力オペレーションで必要とする諸々のデータ管理ルーチンだとか各種制御ブロックがその目的プログラムに組込まれているわけではない. 換言するならば, 実行段階での初期設定が行われるまでは, 完全なプログラムとはいえないのである. この初期設定は, OPEN マクロ命令が実行されることによってはじめて達成される.

OPEN マクロ命令によって実行される OPEN ルーチンは, 指示されたすべての DCB のフィールドを完成したあとで, 各データ・セットの入出力オペレーションに必要なアクセス方式ルーチンをローディングして使用可能な状態



第41図 アクセス方式ルーチンの決定要素

にすると同時に，チャネル・プログラムやバッファ区域あるいは各種制御ブロックを使用可能な状態にしてくれる．

第41図に示したように，アクセス方式ルーチンのどれを選択するかは，DCBの中のつぎのような制御情報を分析することによって決定するのである．

- データの編成法
- バッファ割当て手法
- アクセス手法
- 入出力装置の諸特性

これら情報の組合せで，当該データ・セットの入出力処理にふさわしいアクセス方式ルーチンを決定する．そのあとで，選択したルーチンを記憶するためのスペースを割当てて，当該ルーチンを SYS1. SVCLIB からローディングしてくれるわけである．

OPEN ルーチンでローディングされた各種ルーチン，確保されたチャネル・プログラムやバッファ区域などは，関連する DCB が CLOSE されるまで主記憶装置に貯えられたままの状態にある．

データ・セットに対する入出力オペレーションが完了すると CLOSE マクロ命令を与えることによって，出力バッファに残っているレコードを完全に書き出したうえで，データ・セット・ラベルの検査ないしは作成を行い，該当ボリュームの後始末をする．さらに，そのデータ・セットに対応した DCB を元の状態に復元したうえで，不用になったアクセス方式ルーチンや各種制御ブロックあるいは作業域で占められていた区域および補助記憶装置の未使用部分を解放する．ひとたびデータ・セットが CLOSE されると，それに対応する DCB は元の状態に復元されているからして，その DCB を別のデータ・セットのために用いることもできるのである．

もし，タスク終了以前にデータ・セットを CLOSE しなかったらどうなるであろう．実は，このような場合には，制御プログラムの方で自動的に CLOSE してくれるように，配慮されている．

OPEN マクロ命令にしろ，CLOSE マクロ命令にしろ，1つの命令で2つ以上のデータ・セットの初期設定ないしは事後処理ができる．同時に複数個のデ

ータ・セットを OPEN したり CLOSE したりする方が，別々の命令で行うよりも処理効率が高まるからである．しかし，欠点としては，余分な主記憶スペースがそれぞれのデータ・セットに必要となるので主記憶装置の利用効率は若干さがる．

なお，複数個のボリュームやエクステントで構成されているデータ・セットにおいては，ボリュームやエクステントの最後の位置を感知することによってデータ管理プログラムの EOV ルーチンが自動的に機能してくれる．EOV ルーチンは，データ・セット・ラベルの検査ならびに作成を行ったうえで，ボリュームやエクステントの切換えを行ってくれるものである．

1. 入出力サポート・ルーチンの機能

これまで概観してきたようなデータ・セットに関する初期設定，後始末ならびにボリュームの切換えなどの機能を果すデータ管理プログラムのルーチンを総称して，**入出力サポート・ルーチン**と呼ぶ．入出力サポート・ルーチンは，その機能からつぎの３つに大別される．

- OPEN ルーチン：データ・セットの入出力オペレーションを可能な状態にするための初期設定機能を果す．
- CLOSE ルーチン：データ・セットの入出力オペレーションが完了したあとでの最終処理の機能を果す．
- EOV ルーチン：入出力オペレーションの過程でボリュームの終りやデータ・セットの終りの状態になったときの処理機能を果す．

これら３つの入出力サポート・ルーチンが行ってくれる具体的な機能のうちで主要なものについて詳述しよう．

1） OPEN ルーチンによる初期設定機能

再三指摘してきたように，あるデータ・セットに対応する DCB は，主記憶装置と補助記憶装置間でのデータ転送作業を開始する前に OPEN する必要がある． DCB はプログラムが翻訳される段階で作り上げられるが実行時までは完全なものではないし，またそのデータ・セットのアクセス方式ルーチンや関

連する制御ブロックのいくつかは実行時になってはじめて主記憶装置内に作り上げられる.

OPEN ルーチンが果してくれる主要な機能は，a）基本的初期設定，b）ボリュームの取付けと検査，c）DCB の完成とデータ・セット・ラベルの作成，d）アクセス方式ルーチンの決定，である.

OPEN マクロ命令にコントロールがわたると，b）以降の機能を果すために前準備のための初期設定を行う．前準備としては，OPEN ルーチンが利用する各種作業域の確保と JFCB の読込みがある．OPEN すべき DCB のそれぞれに対して，GETMAIN 命令を使って必要な主記憶装置を確保してくれる．確保する作業域には，DCB を処理するためのもの，JFCB を収容するためのもの，データ・セットの処理のためのもの，OPEN ルーチンがコミュニケーション用に利用するためのものなどがある．これらスペースをあらかじめ確保したうえで，各 DCB に対応する JFCB を SYS1. SYSJOBQE から取り出して確保した JFCB 用のエリアに読込んでくれる．これが基本的初期設定の機能である.

基本的初期設定が終ると，ボリュームの取付け指示と検査を行ってくれる．そのために OPEN ルーチンは，まず，JFCB 内のボリュームに通し番号とそのデータ・セットに割当てられた装置に対応する UCB(Unit Control Block：後述）をチェックすることによってボリュームが取付けられているかどうか検査する．UCB の中にボリューム通し番号があれば，そのボリュームはすでに所定の装置に取付けられていることが確証されたことになる．したがって，オペレータに対して，ボリューム取付けの指示をする必要はない．ボリューム通し番号がなかったら，所定のボリュームをセットする指示を出す.

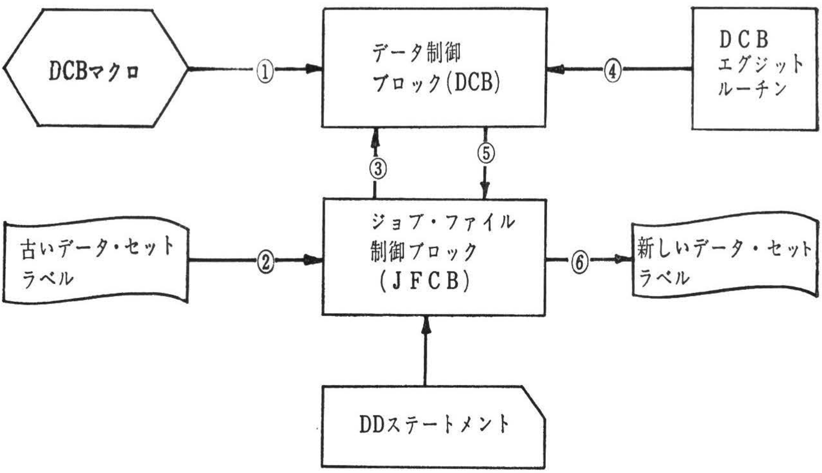
OPENルーチンでボリュームが取付けられていることを確認すると，ボリューム・ラベルを読込んで，その妥当性を検査する．誤ったボリュームが取付けられている場合には，その旨のメッセージを出して別のボリュームを取付けさせたうえで再検査する．検査が終ると，磁気テープの場合には，データ・セット順序番号を使ってテープを動かし，そのデータ・セットの最初に位置づけるようにする.

なお，ボリュームの取付けには，処理すべき最初のボリュームだけをまずセットさせるようなやり方と，該当データ・セットを収容すべきボリュームを一度にセットさせるやり方がある．直接アクセスや索引順次アクセスの場合にはその性質上必ず後者の方法が指示される．

次に「DCB の完成とデータ・セット・ラベルの作成」に移るのであるが，DCB の完成については「データ制御ブロック（DCB）の機能と構成」の項で詳述しておいたので，ここでは今までふれなかったことについて簡単に説明しておこう．

DCB の各フィールドのうちで，OPEN マクロ命令が与えられるまえに情報が貯えられているものの数は，DCB マクロ命令の形式やその中で指定したパラメータの数によって異る．実行時には，空きのフィールドが DD ステートメントの追加情報で満されるのであるが，ジョブ制御ルーチンがこの種の情報を JFCB に収め，その JFCB を使って OPEN ルーチンで DCB に必要な情報を移動してくれる． OPEN ルーチンで行うデータ・セット属性の組合せプロセスには，実はつぎの２通りがある．

●**前方組合せ**（forward merging）：この組合せは，これまでに説明してきた



第42図　DCB に関する前方向組合せと逆方向組合せ

やり方，つまりDSCBまたはデータ・セット・ラベルの情報をJFCBの空きフィールドに組合せ，そのあとでJFCBの情報をDCBの空きフィールドに組合せることによってDCBを完成するステップである（**第42図**の②と③）．データ・セット・ラベルまたはDSCBからJFCBへの組合せでは，JFCBのゼロのフィールド（これが空きのフィールドを意味するのだが）を完成する．この段階は，いうまでもなく入力データ・セットにしか適用されない．一方，JFCBからDCBへの組合せは，OPENすべきすべてのDCBに適用され，DCBのゼロのフィールドがJFCBの情報でうめられる．この時点で，DCBのユーザ・エグジット・ルーチンを使って，DCBの所要フィールドの内容を修正・変更することもできる（第42図の④）．

●**逆方向組合せ**（reverse merging）：この組合せは，DCBからJFCBへ，JFCBからデータ・セット・ラベルまたはDSCBへの方向で進むやり方である（第42図⑤と⑥）．出力の場合だと，DCBからJFCBへの組合せの過程で，JFCBの現存フィールドの内容がDCBの対応するフィールドで置き代る．逆に入力の場合だと，ゼロであったJFCBのフィールドだけがDCBの対応フィールドで組合せられる．JFCBからデータ・セット・ラベルまたはDSCBの書き出しは，出力データ・セットにだけ適用される．

以上のステップが「DCBの完成とデータ・セット・ラベルの作成」機能であるが，OPENルーチンの最後の主要な機能は，「アクセス方式ルーチンの決定」である．**アクセス方式ルーチン**とは，バッファ区域の割当てや制御，バッファ区域からのデータの移動あるいはレコードのブロック化や非ブロック化の機能を果すルーチンのことである．

OPENルーチンは，DCBの中の特定フィールドの情報を使って，そのデータ・セットの入出力オペレーションにふさわしいアクセス方式ルーチンを決定する．そのために，どのルーチンが必要かを決定するための**アクセス方式エグゼキュータ**（access method executor）と呼ばれるルーチンにコントロールを渡す．エグゼキュータでは，そのデータ・セットの入出力オペレーションに必要なアクセス・ルーチンを決定し，そのリストをOPENルーチンに返してくれる．OPENルーチンでは，そのリストを使って所要ルーチンがすでに主記

憶装置にロードされているかどうか確かめ，不在のルーチンに関しては，主記憶装置への呼び込み機能を果してくれる.

以上のような諸機能を果したあとで，はじめてコントロールが処理プログラムに返されるのである.

2) CLOSE ルーチンによる最終処理の機能

データ・セットの処理が最早や必要なくなると，処理プログラムの中では，そのデータ・セットに対して CLOSE マクロ命令を出す必要がある. CLOSE ルーチンでは，TIOT（後述）と呼ばれる制御ブロックを通じて JFCB のアドレスを知る. この JFCB にある情報を用いて，アクセス方式ルーチン，作業域あるいは該当データ・セットに関連する DEB (Data Extent Block：後述) などを解放する. さらには，出力データ・セットのラベルを作りあげたり，DCB を元の状態にもどしたりする機能を果してくれる.

以下で，CLOSE ルーチンが果す機能を処理ステップを踏んで説明してゆこう.

処理プログラムの中で CLOSE マクロ命令が出されると，割込み処理ルーチンを通じて制御権を CLOSE ルーチンに渡す. CLOSE ルーチンでは，まず最初に OPEN ルーチンの場合と同じような基本的初期設定機能を果す.

基本的初期設定のステップでは，GETMAIN マクロ命令を使って，CLOSE ルーチンが使用する各種作業域を確保してくれる. そのあとで，TCB から TIOT の存在場所をつきとめ，さらに TIOT から JFCB のアドレスを得る. その結果，作業域に JFCB および DSCB を読込むのである. また，CLOSE すべきデータ・セットに対してまだ処理が終了していない入出力要求のすべてを解決し，CLOSE の本格的機能を果すことのできる状態にする. これが，CLOSE ルーチンの基本的初期設定の段階である.

次に，トレーラ・ラベルの作成ないしは更新の機能を果してくれる. 磁気テープのデータ・セットだと JFCB 内に入っている情報にもとづいてトレーラ・ラベルが作成できる. 一方，直接アクセス・データ・セットでは，DSCB が読込まれ，そのデータ・セットの最後のブロックとトラックを表示するフィール

ドが更新されるだけである．

出力ラベルの処理が終ると，ボリュームの後始末に必要な処理を行う． CLOSE マクロ命令ないしは DD ステートメントのパラメータで要求されたデータ・セットの処理の取扱いにしたがう． さらには， OPEN ルーチンで DEB の所定フィールドにセットされたマスクを利用して， DCB を元の状態にもどす．つまり，JFCB の情報で満たされた DCB の該当フィールドをゼロの状態にもどすわけである．こうしておくことによって，この DCB が別の処理に再使用が可能な状態になるのである．

DCB の状態を元にもどしたあとは，アクセス方式ルーチン，DCB あるいは OPEN ルーチンで確保した各種作業域を解放し，それらが占有していた主記憶域を他の仕事に利用できるようにする．

このような機能を果たすためのものが CLOSE ルーチンである．なお，タスクが終了した時点でまだ DCB が OPEN されたままの状態であると，つまり，その DCB に対する CLOSE マクロ命令が実行されていない場合には，制御プログラムの方で自動的に OPEN ルーチンに制御権を受渡し，しかるべき処理を行ってくれる．

3) EOV ルーチンの処理機能

EOV ルーチンは，順次編成のデータ・セットのボリュームの終りあるいはデータ・セットの終りの状態に達したときの処理機能を果す．つまり，つぎに示す状態のいずれかが感知されるか，FEOV (Force End of Volume) マクロ命令が出されたとき，自動的にこのルーチンが実行されるのである．

- 入力ボリュームのデータの終りの標識．
- 入力テープ・ボリュームのテープ・マーク．
- 入力直接アクセス・ボリュームのファイル・マーク．
- 出力テープ・ボリュームの End of Reel マーク．
- 出力直接アクセス・ボリュームのエクステントの終り．

なお， FEOV マクロ命令は， 上記のような物理的条件に達しない状態で強制的にそのボリュームの処理を終らせたいような場合に利用する．この場合に

は，別の後続するボリュームが指定されていると自動的なボリュームの切換えが行われる．

ともあれ、上記諸条件において EOV ルーチンが実行されるのである．EOV ルーチンは，まず，標準トレーラ・ラベルのチェックないしは作成機能を果す．そのあとで制御権をユーザー・ラベル処理ルーチンに渡すなり，自動的ボリュームの切換え機能を果すなり，あるいは直接アクセス・ボリュームの場合だと，DD ステートメントの SPACE パラメータで指示されている追加スペースの割当て機能なりを果してくれる．

データ・セットの処理を続行してゆくために追加のボリュームを必要とする場合には，ボリュームの自動的切換えを行ってくれるのであるが，その機能の中には，適正なボリュームが取付けられているかどうかの検証，入力データ・セットの場合のデータ・セット・ラベルの検査，出力データ・セットの場合のラベルの作成などが含まれる．もし，入力データ・セットの終りの理由で EOV の状態が生じた場合には，EOV ルーチンではそのデータ・セットが別のデータ・セットと**連結**（concatenate）されているかどうかを判定する．

連結されているデータ・セットで，かつその属性が同一でない場合には，止むを得ずその EOV の状態にあるデータ・セットを CLOSE せざるを得ない．したがって，EOV ルーチンは，CLOSE ルーチンを主記憶装置の特定エリア（非常駐監視プログラム域）に呼込んで，CLOSE してしまう．CLOSE 機能を果したあとで，OPEN ルーチンに制御権を渡し，連結データ・セットを OPEN する．

一方，連結データ・セットが同一属性をもつ場合には，同一 DCB がそのまま活用できるわけであるので，そのデータ・セットが収容されるボリューム・タイプを決定し，そのボリュームに対しての所要の処理機能を果してくれる．

以上のような機能を果すのが，EOV ルーチンである．

2. 入出力サポート・ルーチンと主なる制御ブロックの相互関係

これまで入出力サポート・ルーチンの機能をみてきたのであるが，さまざまな制御ブロックの情報をよりどころとして，その機能を達成している．そこで，

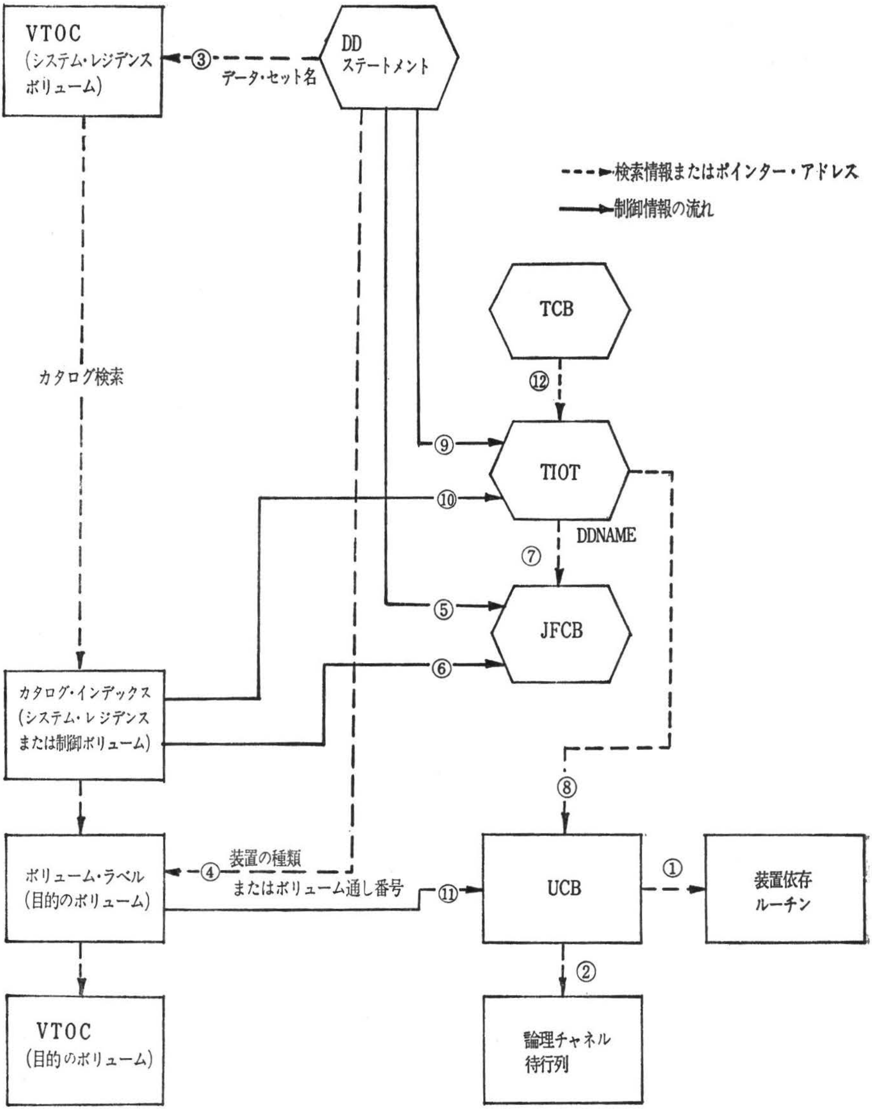
ここでは，入出力サポート・ルーチンがかかわりをもつ各種ブロックの相互関係を中心にその機能をまとめてゆこう．入出力サポート・ルーチンが関係するブロックは，とりもなおさず入出力オペレーションにおいても意味をもってくるものが多いのであるが，OPEN マクロ命令を出す以前に相互関係を保っているブロックと OPEN マクロ命令によってはじめて形成されるブロック，およびその相互機能関係が成立するブロックがある．したがって，OPEN に先立って作り上げられるブロックと OPEN によって生成されるブロックに分けて，その相互関係を説明してゆこう．

1)　OPEN に先立って作りあげられるブロック

第43図は OPEN に先立って作りあげられるブロックとその相互関係を示している．この図を中心に，各ブロックの機能と相互の関係を把えてゆく．

UCB (Unit Control Block) は，そのシステムの全ての入出力装置に対応して存在するものであり，オペレーティング・システムの生成時に各装置に関して指示される情報によって作られる．各 UCB には，装置の状態を示す情報，関係する装置のアドレス，その装置特有の処理を行う入出力監視ルーチン（たとえば，入出力待行列の処理やエラー処理ルーチンなど）のアドレス（①）が入っている．また，その装置に関連して利用される論理チャネル待行列の特定アドレスも入っている(②)．入出力監視プログラムは，この UCB をチェックすることによって，該当装置が "BUSY" かどうかを知る．もし，その装置に対する入出力要請がすぐに処理できない場合には，その要請を一定の形式で待行列の中に入れる．待行列内での待順位は，そのジョブの優先順位によって定められる．そして，待ちにある入出力要請が開始されると，その要請のためのエントリーを指すポインターが UCB に入れられるのである．その結果，割込みが生じると入出力監視プログラムは，UCB を使うことによって，特定の入出力オペレーションを決定することができる．

ところで，DD ステートメントの主要な目標の1つが，データ・セット名を与えることによって，カタログを経由して該当データ・セットを突きとめる（③）ことにあるのはいうまでもない．DD ステートメントでは，この他にも



第43図　OPEN に先立って作りあげられているブロックと相互関係

数多くの情報を指定することができる．カタログに頼らずにデータ・セットを突きとめるには，使用する装置の種類とボリューム通し番号を指定する（④）．ラベルに関する情報，入出力データ・セットのための種々の属性や DASD スペースの指定などもできる．これらの情報の大部分は，ジョブ制御プログラムによって JFCB に収容され（⑤），元の DD ステートメントに代るものとして用いられる．特定のデータ・セットを突きとめるには，装置の種類の表示とボリューム通し番号による方法（④）とデータ・セット名だけを用いる方法（③）があるが，後者の場合だとボリューム通し番号がカタログから引き出されて JFCB に入れられる（⑥）．

各ジョブ・ステップで，ジョブ・スケジューラーは必要な入出力装置の割当てを行う．その割当ての状況を示すために TIOT (Task Input／Output Table) を作る．このテーブルには，各 DD ステートメントごとにエントリーが登録される．各々のエントリーは，データ・セットの DDNAME（⑦），そのデータ・セットに使われる入出力装置を指示する UCB の位置（⑧）などからなる．特定入出力装置をデータ・セットに割当てるための情報は，DD ステートメントを通じて（⑨）またはカタログ・インデックスを通じて（⑩）得る情報とジョブ・スケジュラーが管理する使用可能な装置を示す一覧表とであり，これを突き合せて特定入出力装置を割当てる．そのあとで，該当ジョブ・ステップで必要とする最初のボリュームが所定のボリュームに取付けられているかどうか確かめる．各々のボリュームがセットされると，ボリューム・ラベルを読取って，ボリューム通し番号と VTOC のアドレスとが対応する UCB に入れられる（⑪）．

最後にジョブ・スケジューラーは，そのジョブ・ステップのタスクを作り出す．ジョブ・スケジューラーは，ATTACH マクロ命令を出すことによって，そのジョブ・ステップをタスクとして実行できる状態にする．この過程で，TCB (Task Control Block) が作られる．TCB は，そのタスクの状態を示す情報を集約したものである．状態を示す情報の具体的なものとしては，タスクの優先順位，他のタスクとの依存関係，タスクが実行可能かどうかを示す情報あるいは割込みが生じた時点でそのタスクが用いていた汎用レジスターの内容や

プログラム状況ワード (PSW) の内容などである. タスクの状態を示す情報以外には, そのタスクが用いるプログラムやテーブルのポインター・アドレスが貯えられる. その1つが TIOT のアドレスである (⑫).

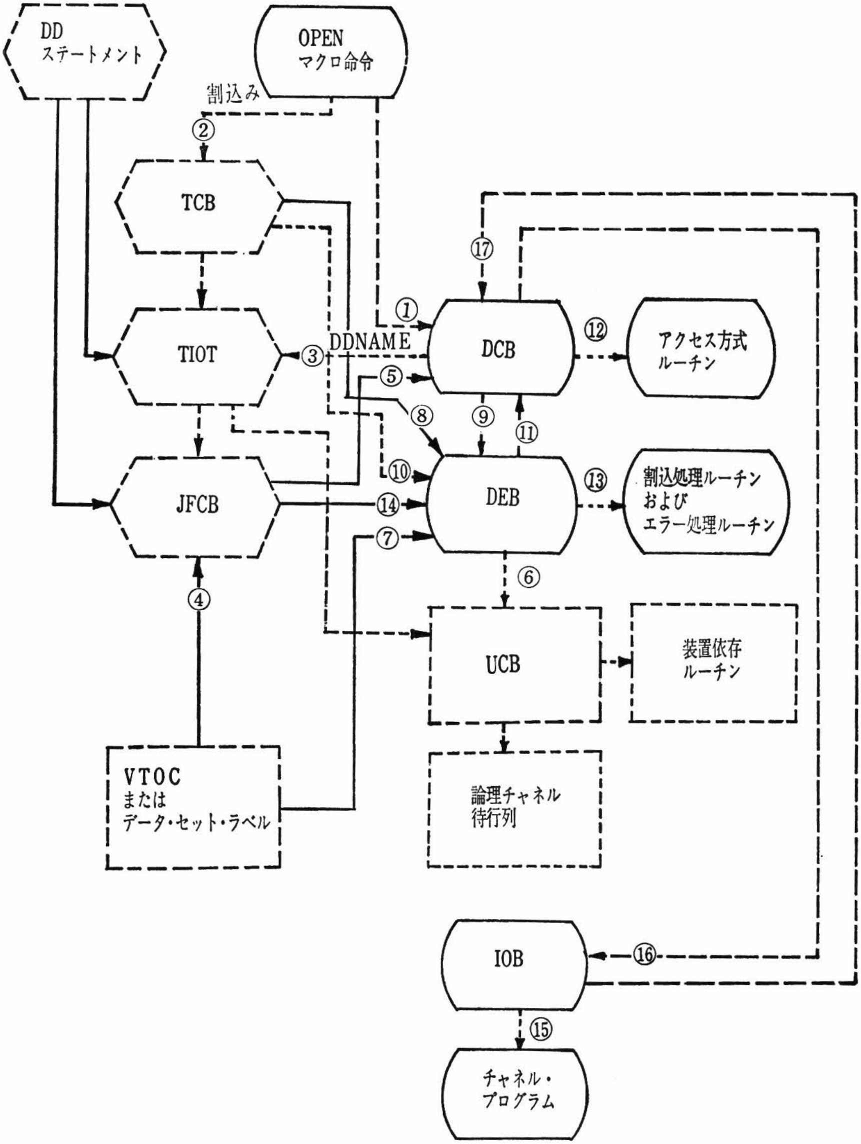
2) OPEN 時に生成されるブロックとその相互関係

OPEN マクロ命令が実行されることによって作りあげられるブロックとその他のブロックの有機的関連づけをしたのが, **第44図**である. OPEN ルーチンで完成されるブロックの主要なものは, DCB と DEB である.

OPEN マクロ命令が実行されると, 完成すべき DCB が指示される (①) と同時に SVC 割込みを通じて TCB のアドレスも示される (②). DCB に収容されている DDNAME を手がかりに, TIOT の中にある処理すべきデータ・セットに応対するエントリーを突きとめる (③). その結果, TIOT を通じて, 関連する JFCB が取出せるのである. JFCB を主記憶装置の作業域に呼込んだあとで, データ・セット・ラベルを読込んで, DD ステートメントでは指定されなかったデータ・セットの種々の属性を JFCB に入れる (④). この時点で, データ・セットの属性とアクセス方式の仕様とがすべて揃ったことになる. そこで, データ・セットの諸属性と処理上のオプションの中で DCB マクロ命令で指示されていないものが JFCB から DCB に渡される (⑤).

つぎに, DEB (Data Extent Block) が作られる. DEB は DCB の情報を追加するためのものであり, 論理的には DCB の延長として考えられる. DEB には, データ・セットの範囲を示す情報 (⑦), そのデータ・セットに関連して用いられるチャネル・コマンドの種類を示すフラグ (⑭), およびタスクの優先順位を示す情報 (⑧) が入れられる. 入出力監視プログラムは, このブロックの情報を用いて, データ・セットの読み書きのさいの妥当な範囲をつかむ. なお, DEB は記憶保護が施された記憶域に置かれるのでその内容を制御プログラム以外で変更することはできない.

DEB は通常 DCB を経由して突きとめられる (⑨) が, TCB を通じて突きとめることもできる (⑩). 後者は, タスクが終了した時点でまだ CLOSE されていない DCB を見出して CLOSE するために必要なのである. また, 順次



第44図　OPEN マクロ命令によって生成されるブロックとその他のブロックの関係

処理の場合には，そのブロックの最初のデータ・セットのアドレスが DEB から DCB に移される (⑪).

DEB が作り出されると，引きつづいて該当データ・セットのアクセス方式ルーチンが SYS1. SVCLIB から選び出され，読込まれる．そして，これらルーチンがローディングされた場所のアドレスが DCB に入れられる (⑫). また，特別な割込み処理ルーチンやエラー処理ルーチンも読込まれ，それぞれのアドレスが DEB に登録される (⑬).

OPEN ルーチンの実行が終了し，該当データ・セットに入出力要請が出されると，アクセス方式ルーチンは目的とするデータ・セットをアクセスするためにチャネル・プログラムと IOB (Input／Output Block) を作りあげる．チャネル・プログラムの各々に対して，IOB が用意される (⑮). IOB とチャネル・プログラムは1対1対で対応しており，チャネル・プログラムは論理的には IOB の1部分とみなすことができる．アクセス方式ルーチンでは，この IOB を調べることによって，該当データ・セットの入出力オペレーションを遂行するためのチャネル・プログラムを決定するのである．IOB は，対応するチャネル・プログラムに関する種々の情報，そのチャネル・プログラムで用いられる ECB (Event Control Block) のポインター・アドレス，DCB のポインター・アドレス (⑯) あるいは，チャネル・プログラムの実行が完了したときの状態，つまり CSW を入れる場所などからなっている．

このようにして，入出力オペレーションに必要な制御ブロックやそれらの相互依存関係が形成されるのである．

十一. データ管理機能のまとめ—入出力オペレーションの流れを中心に

これまで説明してきたように入出力制御の機能では，主記憶装置と補助記憶装置間でのデータ移動に関連した諸々のサービスを提供してくれる．これらサービスを行うのがデータ管理プログラムであり，要約するとつぎの4つの機能

を果しているといえる.

●カタログの管理

●直接アクセス記憶装置のスペース管理

●入出力サポート

●入出力オペレーションの制御

カタログ管理ルーチンは，データ・セットのカタログとそれが収容されているボリュームの維持機能を果してくれる．このルーチンは，カタログされたデータ・セットを突きとめたり，カタログに新たなデータ・セットのインデックスを追加したり，逆に不要なデータ・セットのインデックスを削除したりする.

カタログ管理ルーチンは，SVC ルーチンの１つであり，**非常駐監視プログラム域**（transient area）で働く．ジョブ管理プログラムのうちのジョブ・スケジューラーがこのルーチンを利用してカタログされたデータ・セットを突きとめたり，ジョブ・ステップの実行中に作り上げられたデータ・セットをカタログしたりする．また，ユーティリィティー・プログラムのうちの IEHPROGM でもカタログ管理ルーチンを活用して，カタログの構成要素であるインデックスを追加したり，変更したり，削除したりする．これがカタログ管理ルーチンであった.

直接アクセス装置の**スペース管理**（**DADSM**）**ルーチン**は，直接アクセス・ボリューム上のスペースを割当てたり，不用になったスペースを解放したりする．DADSM ルーチンも，カタログ管理ルーチンと同様に SVC ルーチンであり，非常駐監視プログラム域で働く．このルーチンもジョブ・スケジュラーで利用され，出力データ・セットに対しての補助記憶装置のスペースの割当てを行う．また，入出力サポート・ルーチンでは，追加のスペースを確保したり，未使用のスペースを解放するさいにこのルーチンを利用する.

入出力サポート・ルーチンも SVC ルーチンの一種であり，入出力オペレーションに直結した補足的オペレーションを行ってくれる．これらのオペレーションには，データ制御ブロック（DCB）の Opening 機能，データ制御ブロックの Closing 機能および EOV の処理機能が含まれている．入出力サポート

・ルーチンも非常駐監視プログラム域で機能する．入出力サポート・ルーチンは，処理プログラム，監視プログラムおよびジョブ管理プログラムでも利用される．

入出力オペレーションの制御は，入出力オペレーションが開始されるさいとそのオペレーションが完了するさいに機能する．入出力オペレーションの開始は通常アクセス方式ルーチンと入出力監視プログラムによって遂行される．アクセス方式ルーチンは，問題プログラム域（動的記憶域）で働くが，入出力監視プログラムは，その名が示すとおり，常駐監視プログラム域で働く．一方，入出力オペレーションが完了すると，制御権は入出力監視プログラムに渡る．入出力オペレーションの完了処理の過程では，エラー処理ルーチンやアクセス方式の付加ルーチンが利用され，しかるべき処理が行われる．

以上のような機能を果すのがデータ管理プログラムであったが，ここでは，入出力オペレーションの流れを中心にデータ管理プログラムの機能のまとめを行うことにする．

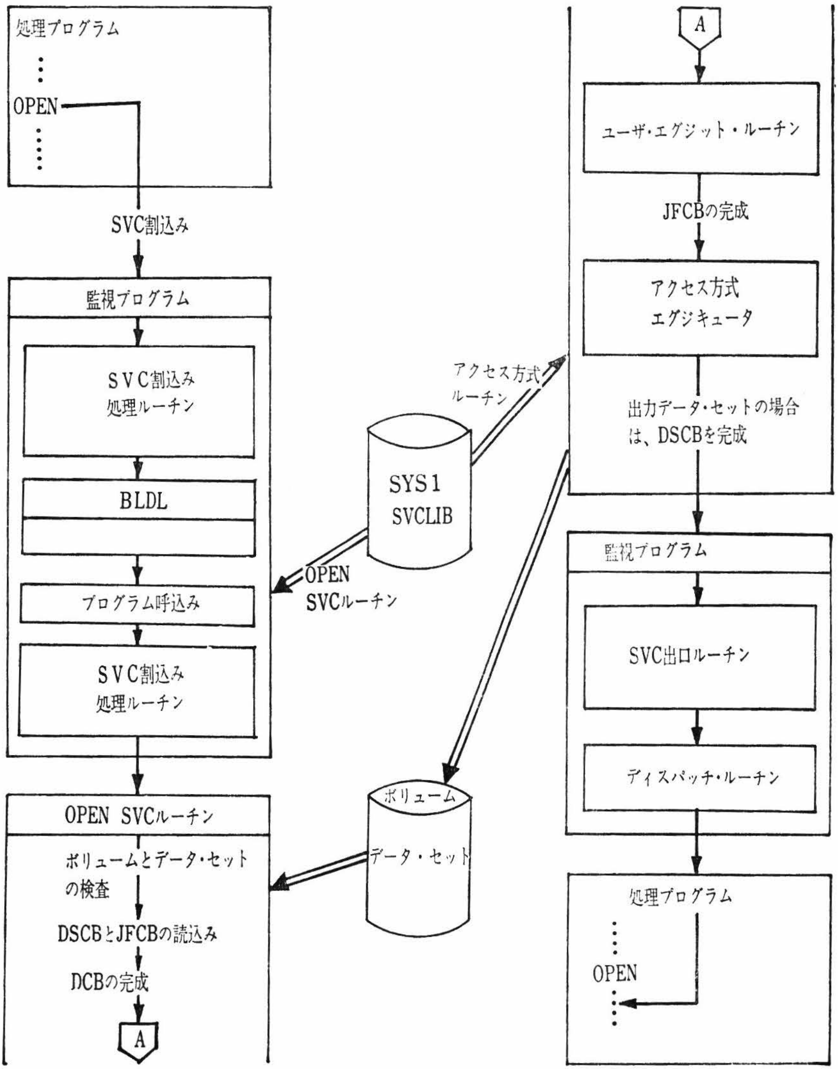
入出力オペレーションに直接関連する処理は，データ制御ブロックがOPENされる時点と実際に入出力オペレーションが実行される時点に行われる．したがって，OPEN 時と入出力オペレーションの実行時に分けて，入出力オペレーションの流れを総括する．

1. OPEN マクロ命令実行時の制御権の流れ

実際にデータ・セットから情報を読みとったり，データ・セットに情報を書き出したりする前に一定の初期化が必要となる．この初期化を行うために使われるのが OPEN マクロ命令であることは再三指摘してきた．

OPENマクロ命令が実行されるさいの制御権の流れを直接アクセス装置上のデータ・セットを例にまとめたのが**第45図**である．他のデータ・セットの場合でも制御権の流れに本質的な差はないので，これにもとづいて OPEN マクロ命令実行時の制御権の流れを説明しよう．

OPEN マクロ命令によって引き出されたルーチンの中には SVC 命令が含まれているので，これが実行されると SVC 割込みが生じる．その結果，CPU の



第45図　OPEN マクロ命令実行時の制御権の流れ

制御権は監視プログラムの中の「**SVC 割込み処理ルーチン**」に移る．このルーチンでは，SVC 割込みの理由を調べたうえで，それに対応する処理ルーチン，つまり「**OPEN SVC ルーチン**」を見つけだすためにデータ管理プログラムの中の BLDL ルーチンを利用する．**BLDL ルーチン**は，ライブラリーに入っている所要のルーチンのアドレスを教えてくれる機能を果す．これによって，SYS1. SVCLIB のどこに OPEN SVC ルーチンがあるかを知ることができる．その結果，SYS1. SVCLIB から，OPEN SVC ルーチンの最初のセグメントを非常駐監視プログラム域に呼込んでくれる．プログラムの呼び込みが終ると，制御権は再び SVC 割込み処理ルーチンに戻され，そのルーチンを通じて「OPEN SVC ルーチン」に制御権が移る．

OPEN SVC ルーチンでは，まず正しいボリュームがセットされていることを確認したうえで，そのボリューム上のデータ・セットをみつけ出す．もし，そのデータ・セットがパスワードによって保護されている場合には，OPEN ルーチンで，所定のパスワードを入力するような要求を出す．オペレータが誤ったパスワードを２回にわたって表示したなら，そのジョブ・ステップの実行は中止されてしまう．正しいパスワードが示されることが確かめられると次のステップに進む．

次のステップでは，動的記憶域に確保した作業域に JFCB と DSCB を読込む．DSCB の情報は，JFCB の中の空きのフィールドをうめるのに使われる．そのあとで，JFCB が DCB の空きのフィールドを充足するのに使われる．DCB が完成し，かつ，DCB の**エグジット・ルーチン**が指定されていれば，そのルーチンに制御権は移る．

「ユーザ・エグジット・ルーチン」の実行が終ると，再び CPU の制御権は OPEN ルーチンに返され，完成した DCB のフィールドから JFCB に情報が移される．つまり，逆方向の組合せが始まるのである．この組合せでは，入力データ・セットの場合だと，JFCB の空きのフィールドだけが，出力データ・セットだと特定のフィールドを除いた全ての関連フィールドが DCB の情報でうめられる．

JFCB が完成されると，OPEN ルーチンの中の「**アクセス方式エグゼキュ**

ータ」と呼ばれるセグメントがローディングされ，それに制御権が渡される．このルーチンは，DEB を作りあげたうえで，必要なアクセス方式ルーチンを選択し，動的記憶域にローディングする．

データ・セットが出力の場合には，JFCB のフィールドが DSCB に組合され，その結果，完全になった DSCB が補助記憶装置上に書き出される．

以上の処理が終ると，OPEN ルーチンは監視プログラムの中の「**SVC 出力ルーチン**」に制御権を返す．このルーチンは，ディスパッチ・ルーチン（制御権委譲先決定ルーチン）に制御権を渡す．**ディスパッチ・ルーチン**ではOPEN マクロ命令による SVC 割込みによって貯えられた PSW を現 PSW にロードすることによって，処理プログラムの中の OPEN 命令のつぎの命令に制御権を戻す．

このような制御権の流れによって，OPENルーチンの機能が達成されるのである．

2.　入出力オペレーションにおける制御権の流れ

入出力オペレーションに関しては，第3章の「入出力オペレーションと監視プログラム」の項でもふれたように，入出力オペレーションを開始させるための処理と入出力オペレーションが終了した時点での処理の2つの局面がある．

入出力オペレーション開始の局面は，処理プログラムの中で入出力マクロ命令が与えられることによって，**アクセス方式ルーチン**にブランチすることで始まる．このアクセス方式ルーチンは，入出力オペレーションを開始するに必要な情報を集め，関連する制御ブロックに集約したうえで，EXCP マクロ命令を出す．EXCP は，SVC 割込みを引き起し，監視プログラムの中の **SVC 割込み処理ルーチン**に制御権を渡す．SVC 割込み処理ルーチンでは，その割込みの理由を分析したうえで，**入出力監視プログラム**に制御権を移す．

入出力監視プログラムは，実際の入出力オペレーションを開始したあと，再び監視プログラムに制御権を委ねる．監視プログラムは，アクセス方式ルーチンを通じて処理プログラムに制御権を戻す．

一方，入出力オペレーションが終了すると，入出力割込みが生ずることによ

ってその旨が知らされる．その結果，制御権は監視プログラムの中の**入出力割込み処理ルーチン**に移る．入出力割込み処理ルーチンは入出力の処理を遂行するために入出力監視プログラムに制御権を渡す．入出力監視プログラムは，特別な処理を行うためにエラー処理ルーチンやアクセス方式の付加ルーチンに制御権を移す．処理が完了すると，監視プログラムを通じて処理プログラムの中断した次の命令に制御権がもどされる．

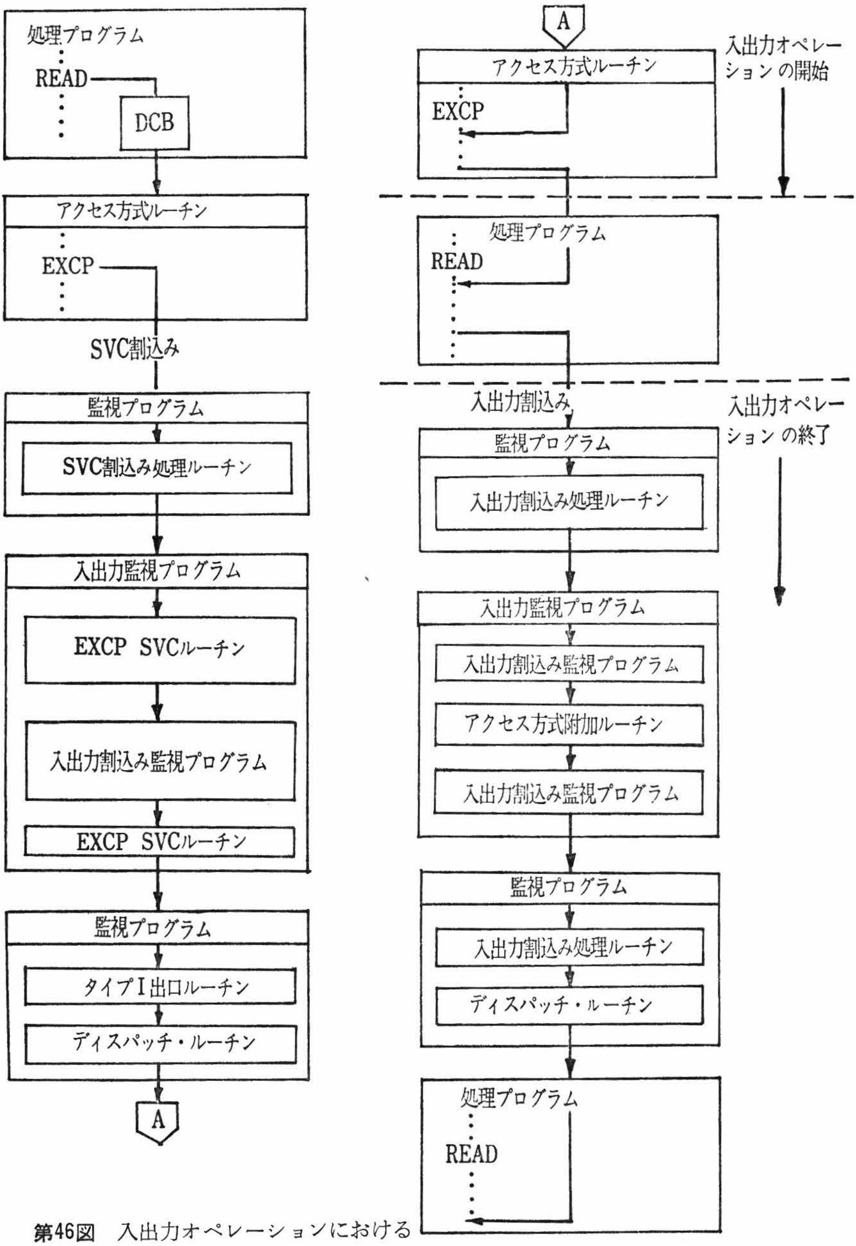
以上が，入出力オペレーションの開始と終了の２つの局面における制御権の流れの概要であるが，以下でもう少し具体的にまとめておこう．そのために入出力オペレーションにおける制御権の流れを，**第46図**に示しておいた．この図では， READ マクロ命令を与えた場合の流れを示している．これにもとづいて説明してゆこう．

入出力オペレーションを実行するためには，そのオペレーションの諸条件に応じたアクセス方式を選択し， DCB マクロ命令の中で規定する必要がある．そのうえ，処理プログラムの中で入出力の要請を出す場合には，アクセス方式に応じた入出力マクロ命令を出す．

待機アクセス方式には， QSAM, QISAM, QTAM などがあり， GET や PUT マクロ命令を使用する．基本アクセス方式だと BSAM, BDAM, BISAM, BPAM および BTAM などがあり，READ や WRITE マクロ命令を使用する．それぞれのアクセス方式に応じた入出力ルーチンが選び出され，入出力オペレーションで利用されるのである．

ともあれ，第46図では READ マクロ命令が与えられた例である．READ マクロ命令が実行されると， DCB が OPEN されたときに DCB の所定の個所に貯えられたアクセス方式ルーチンのアドレスを使って，アクセス方式ルーチンにブランチする．

アクセス方式ルーチンでは，IOB やチャネル・プログラムを作り上げるための場所を確保したうえで，該当する入出力要請に応ずるための IOB，チャネル・プログラムあるいは DECB (Data Event Control Block) を作る． DECB はプログラマーが作る場合もあるが，このブロックには，バッファのアドレス，読取るべきブロックの長さ，該当のデータ・セットに対応する DCB のア



第46図　入出力オペレーションにおける制御権の流れ

ドレスおよび ECB (Event Control Block) などが収容される. アクセス方式ルーチンは, DECBに入っているブロックの長さやバッファのアドレスを使ってチャネル・プログラムを完成するのである. 入出力監視プログラムが必要とする全ての情報をまとめたうえで, アクセス方式ルーチンは, IOBのアドレスを入出力監視プログラムに渡す. これで入出力オペレーションが開始できる状態となったので, EXCP マクロ命令を出すことによって, SVC 割込みを引き起し, 監視プログラムに制御権を委ねる. 監視プログラムの SVC 割込み処理ルーチンは, EXCP による SVC 割込みであることを確認したうえで, 入出力監視プログラムの中の「EXCP SVC ルーチン」に制御権を移す.

EXCP SVC ルーチンでは, 後続する入出力割込み発生を一時延期したうえで, 入出力監視プログラムの一部である「入出力割込み監視プログラム」に制御権を渡す. この入出力割込み監視プログラムでは, 入出力要請の待行列（つまり論理的チャネル待行列）をチェックする. もし, 入出力オペレーションが開始可能なら, この待行列の中で最も優先度の高いオペレーションを開始させる.

このあとで, 入出力監視プログラムは, 再び制御権を EXCP SVC ルーチンにもどすことになる. EXCP SVC ルーチンでは正しく入出力オペレーションが開始されたかどうかチェックしたうえで「タイプ I 出口ルーチン」に制御権を渡す. このルーチンでは, 入出力割込みの延期を解いたうえで, 制御権をディスパッチ・ルーチンに渡す. ディスパッチ・ルーチンは, アクセス方式ルーチンの中の SVC 割込みが生じた次の命令に CPU の制御権を返す.

アクセス方式ルーチンでは, その処理を完了してから制御権を処理プログラムにもどす. READ マクロ命令によって要求された入出力オペレーションは, この時点ではつぎの 3 つの状態のいづれかに置かれている. すでに入出力オペレーションは完了しているか. 目下入出力オペレーションの最中か, または, 装置やチャンネルが "BUSY" であるために入出力待行列におかれているかのいづれかである. 最後の場合だと, 先行する入出力オペレーションの完了によって生じる入出力割込みの処理の直後に開始されることになる. 前二者の場合には, 入出力割込みによって, その完了が知らされる.

入出力オペレーションが終了すると自動的に入出力割込みが生じ，その結果制御権は入出力割込み処理ルーチンを通じて入出力監視プログラムのうちの割込み監視プログラムに移る．割込み監視プログラムでは，入出力割込みがいかなる理由で生じたかを分析し，しかるべき処理を行ったうえで，アクセス方式付加ルーチンへブランチする．このルーチンは，入出力オペレーションに使われているアクセス方法に固有な処理を行う．そのあとで再度制御権が入出力監視プログラムに渡り，ここで，入出力待行列におかれている入出力オペレーションの要求をスタートさせるのである．そののちに，入出力割込み処理ルーチンを通じてディスパッチ・ルーチンに制御権を返す．ディスパッチ・ルーチンは，処理プログラムの中の入出力割込みが生じた次の命令に制御権をもどすのである．

以上のような制御権の流れにもとづいて入出力オペレーションが達成されてゆくのである．

本章で説明してきたように，データ管理プログラムは，さまざまなデータ・セットを統一的手法で管理し，その識別や記憶あるいは取出しを行ってくれる．直接アクセス装置のスペースの割当てというわずらわしい仕事を代行してくれる．また，データ・セットの編成やバッファリングあるいはデータの転送などに関しても種々の手法を提供してくれる．そのうえ，プログラミングの効率を高め，入出力装置の利用率を高めるために，実際の装置から独立して原始プログラムが書けるような便宜をも提供してくれる．このような機能によってオペレーティング・システムは，さまざまな構成のハードウェア，さまざまなデータ処理業務，さまざまなデータ処理形態の要求に比較的容易に応ずることができるのである．

なお，本章ではデータ管理プログラムの機能のうちで，「データ・セットの編成法とアクセス手法」について詳しく言及できなかったが，章をあらためて詳述することにする．

索　　引

あ　行

か　行

さ　行

た　行

な　行

は　行

ま　行

や　行

ら　行

わ　行

<著者略歴>

え むら じゅん ろう
江 村 潤 朗

昭和13年	広島県三和町にて生まれる
昭和37年	早稲田大学政治経済学部卒業
同年4月	日本アイ・ビー・エム株式会社に入社．カストマー教育のインストラクターを約5年間担当．昭和43年よりコンピュータ教育の手法や教材の開発にあたる
現在	カストマー教育・企画管理担当マネジャー
著書	「すぐに役立つコンピュータ入門」（日本工業新聞社）
主要論文	「企業内におけるコンピュータ要員教育——その問題点と将来の方向——」（情報処理 Vol. 12, No. 11 (1971) p. 731〜734）その他教育工学に関するもの，企業内コンピュータ教育の進め方に関するもの，プログラマーの適性に関するものおよびコンピュータのハードウェアとソフトウェアに関するものなどがある
現住所	川崎市高津区向ケ丘 430 宮前平グリーンハイツ35—104

OPERATING SYSTEMS Vol. 1

JUNRO EMURA
International Business Machines Corporation of Japan

Published by Nippon Computer Kyokai

Printed in Japan.

オペレーティング・システムへの構造的アプローチ（上巻）

昭和47年7月1日	初版発行
昭和47年7月30日	2版発行
昭和47年8月11日	3版発行
昭和48年3月10日	4版発行
昭和48年11月30日	5版発行
昭和49年12月25日	6版発行
昭和50年11月15日	7版発行
昭和51年11月15日	8版発行
昭和52年9月15日	**9版発行**

著　者　江　村　潤　朗
発行者　村　越　勝　弘
編　集　**日本コンピュータ協会**
発行所　**科学技術出版社**
東京都文京区本郷2丁目15—13
電話 東京 (03) 813—6920

乱丁本・落丁本はお買求め書店にてお取りかえいたします.